Pentium® Pro Processor BIOS Writer's Guide

Version 2.01

February, 1996

Information in this document is provided solely to enable use of Intel products. Intel assumes no liability whatsoever, including infringement of any patent or copyright, for sale and use of Intel products except as provided in Intel's Terms and Conditions of Sale for such products.

Intel retains the right to make changes to these specifications at any time, without notice.

Contact your local Intel sales office or your distributor to obtain the latest specifications before placing your product order.

MDS is an ordering code only and is not used as a product name or trademark of Intel Corporation.

Intel Corporation and Intel's FASTPATH are not affiliated with Kinetics, a division of Excelan, Inc. or its FASTPATH trademark or products.

* Other brands and names are the property of their respective owners.

printed on
recycled paper

Revision History

This revision accompanies the source code modules that have been tested on an Intel validation platform.

The document has been updated to reflect the latest Pentium Pro processor silicon.

Revision	Description	Date
0.9	Pre-silicon draft for limited distribution.	10/94
2.0	Current release.	01/96
2.01	Updates to Chapter 8.	02/96

Table of Contents

Figures

Tables

1 Introduction

This document explains BIOS programming for systems based on the Pentium® Pro processor. The document may accompany Intel Pentium Pro processor Basic Input Output System (BIOS) source code example files, which serve as the Pentium Pro processor and 82450 PCIset reference BIOS implementation. This implementation targets a typical system based on the Pentium Pro processor-82450 PCIset. See the Related Documents and Products section for ordering information, if you need the source code files.

1.1 Purpose

The *Pentium Pro Processor BIOS Writer's Guide* describes the issues that surfaced during the design and development of Intel's Pentium Pro processor reference BIOS system code. Its purpose is to provide OEMs and BIOS developers with the example source code and documentation needed to address these issues while developing a system based upon the Pentium Pro processor. This document is a supplement to other manuals for the Pentium Pro processor and 82450 PCIset.

1.2 Target Audience

This document is intended for Pentium Pro processor BIOS architects, code developers and others interested in the software issues of a Pentium Pro processor-82450 PCIset based system.

1.3 Related Documents and Products

This document refers to the following publications and products:

- *Pentium Pro Operating System Writer's Guide*, order number 2422692-001, Intel Corporation.
- *MultiProcessor Specification*, v1.4, order number 242016-004, Intel Corporation.
- *PCI BIOS Specification*, Revision 21, August 26, 1994, Intel Corporation.
- *Pentium Pro Processor BIOS Reference Kit, Reference Source Code V2.0*, available late February, 1996. Order using the following telephone numbers:
  U.S.A./Canada 1-800-253-3696 Option 1
  International 1-503-264-2203 Option 1

1.4 General Concerns of Modularity

The source files that accompany this document manage Pentium Pro processor components, the 82450 PCIset and some of the platform resources. These CPU and chipset initialization files might differ in architecture when compared to similar initialization files in a particular core BIOS.

The source file architecture is based on following key design decisions:

- The code is structured so that it can be easily hooked into any standard BIOS.
- Chipset and other register information is easily extracted from the source files. The implementation contains no complicated, implicit data structures for register values, but uses simple read/write/alter macros, so that one can easily determine values loaded into the chipset register and utilize the information in any form that is desirable for a particular BIOS.

The following implementation strategies achieve these two design goals:

- A single Register Initialization Hook contains all the pre-initialization for the chipset and all the system and platform components that are supported. This code block must be hooked into a typical core BIOS in a very early Power On Self Test (POST) stage, before memory-sizing and AutoScan. The code in this module is written such that it does not use any system memory, so that one can safely execute the code prior to memory configuration.
- System and Video shadow routines have been isolated as separate hooks to allow incorporation of these routines into different BIOS systems in a flexible manner.
- All the advanced initialization of the chipset is grouped into a separate group of routines. This code can and should be hooked at a much later stage in POST, when memory is available.
- The AutoScan module for the 82450 PCIset, including code for switching the machine to protected mode, is isolated in a separate macro that can be called as an independent module.
- Simple register read/write/alter macros are used so that the reader can easily extract the information about the register data. For example, a register of the 82450 PCIset set might be manipulated as:

```
WriteOPB_C_8BitRegister  OPBMemoryATTR0,Data
or
AlterOMC8BitRegister     OMCMemoryATTR0,Data1,Data2
```

Where:

Data1 specifies the AND pattern of bits for the register under consideration, and

Data2 specifies OR pattern of bits.

So, if a particular Pentium Pro processor BIOS must conform to a specific register table architecture, the register information can easily be extracted to build the table.

1.5 Contents of the Source Files

The following list provides a brief description of the contents of the Reference BIOS source files.

Files Contained in REFBIOS Directory

`orion.inc`	initialization code for 82450 PCIset.
`orion.asm`	procedures that support the macros located in the initialization code.
`reset.inc`	procedures that are executed every time a CPU reset occurs.
`p6access.mac`	macros that support access to Pentium Pro processor registers.
`p6_equ.inc`	equates for Pentium Pro processor registers.
`p6ftsfar.asm`	code that initializes the Pentium Pro processor.
`orion.mac`	macros that access the 82450 PCIset components.
`orionequ.inc`	equates for the 82450 PCIset.
`system.mac`	PC-AT* macros for supporting various PC-AT functions.
`sys_equ.inc`	equates that support the `system.mac` file.
`ptform.mac`	macros that support the necessary hooks of the Pentium Pro processor platform.
`pci.mac`	PCI-Config 1 type access macros.
`ptform.inc`	equates needed to manage the Pentium Pro processor platform.
`cntrl.inc`	option flags that control the Pentium Pro processor BIOS build.
`biosgd.doc`	a copy of this document in Microsoft* Word* 6.0 format.
`p6mp.asm`	Pentium Pro processor MP initialization code.
`p6mpfar.asm`	Pentium Pro processor MP Support code.
`mem64meg.asm`	Code to support the call interface that manages memory systems larger than 64 megabytes.
`P6int15.asm`	Code to support INT 15 API for BIOS Update feature.
`update.asm`	Code that loads BIOS Update Data.
`flshrdwt.asm`	Code to support read/write into Intel Flash device.

Files Contained in REGEDIT Directory

`p6h.zip`	a package that contains executables and support files for the Pentium Pro processor and 82450 PCIset set register editor Program.

2
Initialization of the 82450 PCIset

This chapter explains the initialization mechanisms used in the reference BIOS implementation. The 82450 PCIset is used in the sample implementation. The same general principles can be applied to other chipsets.

2.1 An Overview of the Target Platform

The assumed target platform has two or four Pentium Pro processor CPUs with dual PCI bridges that provide a twin PCI bus architecture. The two PCI channels are peer channels at the host level. Figure 2-1 provides the block diagram of the target system.

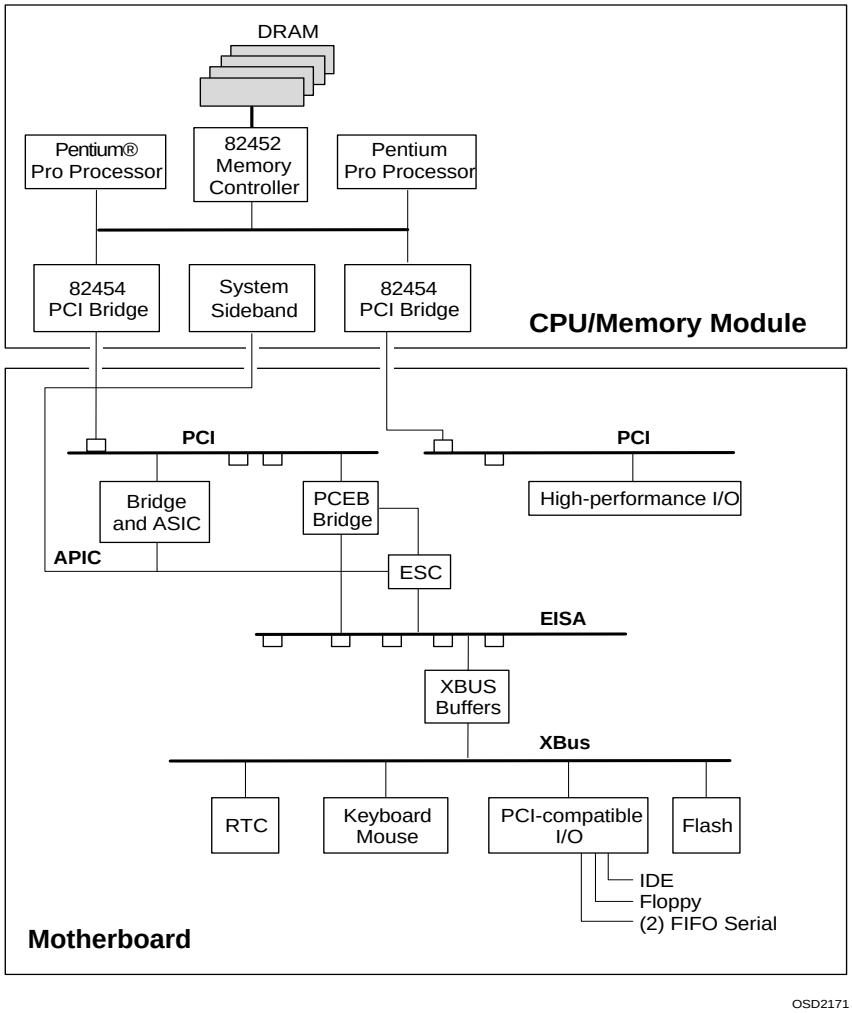

Figure 2-1. Target Platform for Reference BIOS Implementation

PCI and PC compatibility is derived from the PCI bridge chipset and the motherboard. The two 82454 PCI bridges in the CPU/Memory module are connected into the two PCI buses of the motherboard. The first bridge that connects to the PCI bridge chipset and branches to EISA/ISA bus systems is called the compatibility bridge, because all PC compatibility devices, such as DMA and the Key Board Controller, are located on this channel. The second bridge produces a second PCI bus to which high performance I/O components can be attached. These I/O components on the second PCI bus provide another high bandwidth I/O channel that can meet the high-performance requirements of the Pentium Pro processor.

The PCI bridge chip is mainly used to extend basic interrupt support and to give full interrupt steering ability to the PCI channels. In addition, the motherboard must supply at least one I/O APIC so that multiple Pentium Pro processor CPUs can function in a way that complies with the *MultiProcessor Specification*. The 82452 Memory Controller is located on the CPU/Memory module and supports the DRAM modules. The compatibility channel also accommodates PC-compatible hardware that supports a parallel port, IDE hard disks, two serial ports, and a floppy controller.

2.2 General Issues of a PCI BIOS

The PCI specification presents two distinct methods of accessing the registers contained in the PCI components. These methods are called PCI Configuration Mechanism #1 and PCI Configuration Mechanism #2. Most PCIsets that Intel has designed and produced for i486™ and Pentium processors support access using Configuration Mechanism #2, although a few support access by Configuration Mechanism #1. Configuration Mechanism #2 is not a preferred mechanism, although at first glance it appears easier to implement and requires less code space.

The 82450 PCIset supports only Configuration Mechanism #1 due to the following reasons:

- Configuration Mechanism #2 uses a separate bridge-forward register to address the PCI components. This consumes the entire CXXX I/O space.
- The hardware designer cannot use the same PCI components at the same PCI bus level. In other words, two 82454 PCI bridges can not be directly connected to the Pentium Pro processor bus. This precludes providing multiple PCI buses and memory controllers at the same PCI bus level, which is required to effectively utilize the full performance potential of the Pentium Pro processor.

2.3 PCI Configuration Mechanism #1 Access Macros

This section describes the design issues confronted when implementing the Configuration Mechanism #1 PCI access method. In Configuration Mechanism #1, data is read and written as 32-bit chunks. The access mechanism is initiated by writing the address of the device with proper bus, device, and register numbers into `0CF8h` as a 32-bit quantity. The format of this 32 bit quantity includes six bits (bit 2 through bit 7) that describe the register number. The data can be read back from the `0CFCh` address as a 32, 16 or 8 bit quantity. In Intel architecture, data that is read as 32 bits from an I/O port spans four I/O addresses to preserve byte addressability, which is very important for compatibility.

Given these architectural attributes, following are two typical ways that Configuration Mechanism #1 access routines can be implemented:

- Compute the address of the data register by taking into account the offset of the given PCI register from the nearest low 4-byte boundary. For example, to read/write the PCI register at `50h` as an 8-bit quantity, choose the data register `0CFCh`. For the register at `51h`, choose the data address `0CFDh`, for `52h` choose `0CFEh`, and for `53h` choose `0CFFh`.
- Design the Configuration Mechanism #1 PCI access macro code to locate and shift data bits properly, depending on the address boundary of the particular PCI register being read. If address `10h` is read as a 32-bit I/O read operation, the 8 data bits of I/O port `10h` are placed in positions D0 through D7, the 8 data bits of I/O port `11h` are placed in positions D8 through D15, and so forth. Thus, Configuration Mechanism #1 PCI access macro code must be able to locate and shift data bits properly, depending on the address boundary of the particular PCI register being read. The reference BIOS implementation uses this method.

For either approach, the 4-address span makes Configuration Mechanism #1 access macros longer than Configuration Mechanism #2 access macros.

Follow these steps to minimize the impact of code space when using the Configuration Mechanism #1 PCI access code in the Reference BIOS:

- Always access the PCI register manipulation code as a call from POST modules wherever some memory is available.
- In cases where no memory is available, execute the PCI register manipulation code by transferring the control to it through a jump instruction and passing a return address pointer in one of the registers. For an example, see the Register Initialization Hook code in the `orion.inc` file. The disadvantage of this method is that working registers become unavailable because they are used to pass both the parameters and the return address information.

Once you have chosen to use PCI Configuration Mechanism #1, all access to the devices on the platform must be performed using the same mechanism, including all PCI devices that are embedded on the motherboard and any PCI devices that occupy PCI slots. Partial use of access Configuration Mechanism #1 in a given platform is illegal according to the *PCI Specification.*

2.4 Initialization of 82454 PCI Bridges

In the Pentium Pro processor-based host platform, two PCI busses are generated by two 82454 PCI bridges that are connected to the Pentium Pro processor system bus. These two 82454 PCI bridges are peers at the host level. Initialization of 82454 PCI bridges occurs at three stages:

- Early POST initialization
- PCI resource allocation
- Advanced 82450 PCI chipset feature initialization

2.4.1 Early POST Initialization of 82454 PCI Bridges

Early POST code through the OrionRegInitHook macro initializes a few fundamental registers. This code also detects and identifies 82454 PCI bridges before attempting to initialize them.

A configuration cycle that has a bus number equal to 0 enables access to the internal registers of both bridges. The bus number of the second bridge, also called the non-compatibility bridge, is initialized to an arbitrary selection of `80h` during this initialization process. PCI slots under the second bridge have the same device numbers as those in the slots under the compatibility bridge and are only distinguished by their different bus number. The distinction between identically device-numbered slots under both bridges is achieved by programming a different PCI bus number for the second bridge. Then, the PCI devices under the second bridge can be accessed with this new bus number, `80h` in the Reference BIOS.

NOTE. *Even after a new bus number is programmed for the second 82454 PCI bridge, the internal registers of the second bridge can still be accessed with bus number 0 and the appropriate second bridge device number.*

The 82454 PCI bridge has been designed with multiple PCI channels in mind. The second bridge "knows" that it is a non-compatibility channel chip and initializes itself with different power-on default values compared to the compatibility bridge. This distinction is made at the time of power on reset by sampling the hardware state of certain 82454 PCI bridge pins, simplifying the actual early POST initialization modules. If the bridge did not have this feature, a number of second bridge registers would have to be initialized to avoid potential resource conflict. For example, video memory ranges, ISA bus I/O ranges, BIOS address ranges, and so forth, must be configured on the second bridge as disabled, so that the second bridge does not respond and corrupt the valid data that the compatibility bridge brings onto the Pentium Pro processor bus in these ranges.

2.4.2 PCI Resource Allocation on 82454 PCI Bridges

PCI resource allocators call some of the initialization modules for 82454 PCI bridges in order to initialize both compatibility and non-compatibility bus resources. This hook is built into PCI bus scan code so that it works in a twin PCI bus architecture. Source code modules in the reference BIOS implementation contain procedures to program the PCI resources into 82454 PCI bridge registers, but the core PCI base BIOS must provide a hook to call the chipset-dependent routines, as explained in greater detail below.

The PCI bus initialization code modules search for PCI devices on both compatibility and secondary PCI bus structures. Consider a situation in which these routines have found a PCI device residing on the second bridge's PCI bus and that device requires an I/O range and a memory range. In this situation, the BIOS must be able to take these resource requests and program them into both 82454 PCI bridges. In fact, the compatibility bridge must be programmed in such a way that it does not respond to these ranges, but the secondary bridge must be programmed so that it does respond to these ranges. Otherwise, data on the Pentium Pro processor bus is corrupted whenever this PCI device is accessed. I/O ranges can be programmed using `OPBIO_Range` registers and memory ranges can be enabled and disabled using `Memory_Gap` registers. For example, if PCI resource allocators want to channel I/O addresses `8000h-8200h`

down to the second PCI bus, then program the same start (8000h) and end (8200h) addresses to both first and second 82454 bridge I/O gap range registers (Offset 98h), set the enable bit (bit 31 of the gap range register) in the second bridge, and reset the same bit in the first bridge. This tells the first bridge not to respond to this I/O range (8000h-8200h in this example) and tells the second bridge to respond to the same I/O range. Note that a compatibility 82454 PCI bridge responds to all addresses by default and must be told only about the range of addresses that it must not respond to. The same explanation holds for memory gap range registers at offset 88h that are used to allocate PCI memory address ranges.

2.4.3 Advanced 82450 PCIset Feature Initialization

This initialization is usually done at an advanced POST stage. In the sample code, the NewFeaturesHook module is called during late POST. Memory and stack space are available at this stage, so requirements on the initialization code are less stringent at this time. This code module consults CMOS-based setup to determine features that should be enabled. The 82450 PCIset offers a rich set of features that customers can use depending on the platform support. Some of these new features, listed below, need special processing while programming. For a complete list of the features, refer to the code module called NewFeaturesHook in the reference code packet.

82452 Memory Controller System Error Reporting

Enabling single bit error correcting of Pentium Pro processor data must be done before clearing out memory during a memory scan. This feature is enabled by setting bit 9 of the 82452 memory controller system error reporting command early in the chipset initialization.

82452 Memory Controller Address Bit Permuting

Enabling this feature changes the way the memory pages are addressed, therefore it must be enabled before the BIOS starts using memory. Also, the BIOS must ensure the number of memory rows is a power of two, all rows are the same size, and all populated rows are adjacent and start at row 0. If any one of these conditions is not satisfied, this feature must not be enabled. In the Pentium Pro processor BIOS reference code, this feature is enabled by the AutoScan module.

2.5 82452 Memory Controllers

The 82452 controller is a versatile multi-interleave memory controller that supports very large DRAM systems. Its register organization enables the chaining of multiple 82452 memory controllers if needed in very large systems. The second controller can be programmed to start refreshing DRAMs from the top of the memory of the first controller. Use of Configuration Mechanism #1 PCI access enables use of multiple memory controllers on the same Pentium Pro processor bus. Although some of Intel's internal evaluation systems support such a second 82452 memory controller, the Reference BIOS makes no attempt to do so.

Initialization of the 82452 memory controller falls into 2 stages:

- Early POST initialization: During this stage the memory mappings need to be returned to disabled if the reset is not due to a hardware reset.
- AutoScan module initialization : An AutoScan module determines the size of the onboard memory and adjusts the memory size registers to support them. After running the AutoScan module and setting up memory, advanced features of the 82452 memory controller can be enabled based on the user driven Setup menu.

2.5.1 Early POST Initialization of the 82452 Memory Controller

Not much initialization needs to be done to the 82452 memory controller in early POST because the default values are sufficient to begin AutoScan. The controller memory timings must be adjusted to the slowest values to make sure that the AutoScan module runs properly, then the memory must be adjusted to run at whatever speed it is designed to operate. If the system contains a second controller, you must disable it so the AutoScan routine can deal with the first controller before any others.

2.5.2 AutoScan Module Initialization for the Memory Controller

The AutoScan module must find the size and shape of the memory and adjust the 82452 memory controller registers accordingly. The AutoScan algorithm also finds the best possible interleave for the installed memory subsystem. Before starting the actual algorithm that sizes the memory, initialization code must perform the following actions:

- Pentium Pro processor caching must be turned off. In the sample code, caching is disabled by running an early POST RegisterInitHook routine. The Pentium Pro processor cache can be turned off using the conventional CD bit that is present in the `CR0` register. Consult the `system.mac` file in the Reference BIOS for more details.
- The AutoScan algorithm needs all types of paging mechanisms and memory-mapping schemes to be disabled, including any SM memory mappings, gap registers, and any other such features.
- The BIOS must switch the processor to protected mode with `A20` of the processor enabled so that it can access all the physical memory located on the bus. AutoScan software does not have to support asymmetric memory types since the 82452 memory controller hardware supports them automatically.

2.5.3 AutoScan Algorithm

This algorithm supports the common Column Address Strobe (COMCAS) feature of the 82450 PCIset, a feature that is needed to support double-sided byte parity SIMMs.

The algorithm follows this 8-step sequence to determine the memory interleave that is supported by the motherboard's memory system:

The first four steps determine whether to set the COMCAS bit.

1. Set up the 82452 memory controller command register to 1-to-1 interleave, with the COMCAS enable bit (Bit 9 of the 82452 memory controller Command register) turned off.
2. Set the DRAMRowLimit0 register to `0001h` and other row registers (DRAMRowLimit1 through DRAMRowLimit7) to a maximum, `07ffh`.
3. Write 32 bytes of known pattern of data from address `0h` onwards.

4. Read back the 32 bytes starting from address 0h and analyze the results, as indicated in this example of a 32-byte pattern that is visualized as eight 4-byte chunks of data:
   — If matching chunks with the original data pattern written to address 0h are organized as adjacent pairs of 4-byte chunks, leave the COMCAS enable bit in default setting and go to Step 5 to determine interleave of the system.
   — If one 4-byte chunk matches the original data that was written but its adjacent 4-byte chunk does not match, turn on the COMCAS bit then repeat steps 3 and 4. If a match is found between adjacent 4-byte chunks this time, leave the COMCAS bit set and proceed to Step 5 to determine the interleave of the system. If no match is found for any combination of COMCAS bit settings, then the memory system is faulty and/or does not contain memory in row 0. This is a terminal error and in this case the system must be halted.

The next three steps determine the interleave bits.

5. Set up the 82452 memory controller command register to reflect 4:1 interleave. In some systems burst delay may have to be set to 0 for 4:1 interleave configuration.
6. Write 32 bytes of data with a known pattern from address 0h onwards.
7. Read back the 32 bytes of this know-pattern of data from address 0h and analyze the results. In this example a 32-byte pattern is visualized as four 8-byte chunks of data:
   — If all four 8-byte chunks, the full 32 bytes of data, are intact, then the memory subsystem has 4-to-1 interleave. Set the 82452 memory controller command register to reflect 4:1 interleave.
   — If any two 8-byte chunks are intact, then those two groups could have 2-to-1 interleave. Set interleave bits 11 to 14 (in 82452 Command Register at offset 4Ch), as shown in Table 2-1, and set bits 3 and 4 to indicate 2-to-1 interleave.

NOTE. *At least 2 bits in this table must be set for 2:1 interleave to operate.*

   — If no two blocks of 8-byte chunks are intact, only 1-to-1 interleave is possible in this memory system. Set the interleave bits to indicate the active interleave as shown in Table 2-1.

Table 2-1. Interleave Bit Settings in 82452 Command Register (Offset 4Ch)

Bit number	is set to indicate data is OK in the
11	first 8-byte chunk
12	second 8-byte chunk
13	third 8-byte chunk
14	fourth 8 byte chunk

   — If a single 8-byte chunk is not intact, ROW 0 contains no valid memory. Indicate a memory failure and halt the system.

The last step determines the memory size.

8. Set the DRAM row limit registers successively to reflect the memory sizes in Table 2-2:

Table 2-2. DRAM Rowlimit Settings

Rowlimit Registers	Interleave Size
8M, 16M, 32M, 64M, 128M	1:1 memory
16M, 32M, 64M, 128M, 256M	2:1 memory
32M, 64M, 128M, 256M, 512M	4:1 memory

For each of the row limits above, write data to each 4 megabyte boundary (e.g. 4 megabyte, 8 megabyte, etc.) and check for an alias to address 0. The actual size of the installed row is the previous 4 megabyte boundary size that was successfully tried (i.e., no alias to address zero was found).

2.5.3.1 Sizing Algorithm Implementation Details

There are several ways to implement the sizing (step 8) algorithm. The implementation in the reference BIOS is described in the steps below.

1. Starting with DRAM row 0, sequence the row limit using the values in Table 2-2, while testing for memory aliasing. To test for aliasing, write data patterns to addresses in 4 megabyte increments starting at the 4 megabyte address. Check for an alias to address 0 each time. Once aliasing occurs, the size of row 0 equals the last successful row setting. For example, if the row limit was set to 64 megabytes when aliasing occurred, then the size of row 0 is 32 megabytes.

   For a PC hardware compatible design, Row 0 must contain some memory.

2. Assuming that row 0 has been sized to X megabytes and the interleave of the memory subsystem is equal to IL (where IL is equal to 1 for a 1:1 memory subsystem, 2 for a 2:1 memory subsystem, and 4 for a 4:1 memory subsystem), program the row limit registers as shown in Table 2-3.

Table 2-3. DRAM Rowlimit Settings

Rowlimit Registers	Size of Memory
DramRowLimit0	= (X/4)
DramRowLimit1 through 7	= (X/4) + (IL)*(8/4)

2.5.3.1 Blank Row Management

To verify that memory is installed in row 1, write to address X+4 megabytes and check if there is any memory present. If there is no memory at this address, row 1 is a blank row. Copy the contents of DRAM row 0 memory size register to row 1 and continue on to size the next row. In this case, if there is no memory in row 1, the registers are set as in Table 2-4.

Table 2-4. Setting Rowlimits with a Blank Row (Ex: Row 1 is a blank row)

Rowlimit Registers	Size of Memory
DramRowLimit0	= (X/4)
DramRowLimit1	= (X/4)
DramRowLimit2 through 7	= (X/4) + (IL)*(8/4)

Once blank row management is complete, continue writing at addresses of 4 megabyte increments, starting from X megabytes, while checking for aliases to the X megabyte address. If aliasing does not occur until (X+(IL)*8/4), set the row limits as shown in Table 2-5 and repeat the procedure until an alias is detected.

Table 2-5. Setting Rowlimits When Row 1 is a Blank Row

Rowlimit Registers	Size of Memory
DramRowLimit0	= (X/4)
DramRowLimit1	= (X/4)
DramRowLimit2 through 7	= (X/4) + (IL)*(16/4)

Upon alias detection, the size of the row equals the last successful row size setting.

2.5.3.2 General Description of Sizing the "Nth" Row

This section provides a general description of how to size a row.

If the Nth row is to be sized and the sum of all previously sized rows is Y megabytes, then:

- DramRowLimit(0) through DramRowLimit(n-1) are set to their proper sizes, taking into consideration any blank rows.
- DramRowLimit(n) through 7 [Where N <= 7] = (Y/4)+ (IL)*(8/4)
- Perform blank row management, as described earlier, by checking for memory at (Y+4) megabytes.

After blank row management, start at Y megabytes and continue writing at address increments of 4 megabytes. Check for aliases at address Y megabytes. If aliasing does not occur until (Y/4)+(IL)*(8/4), increase the row limits to the next setting and repeat the procedure until an alias is detected. The size of the Nth row equals the last successful row size setting.

2.5.3.3 Programming Issues of the AutoScan Algorithm

The AutoScan algorithm implementation contains the following traditional programming complexities:

- As the algorithm is coded, more and more registers are used. Eventually, there are few registers left to call the register access code. Memory is still not available, so calling the register access code is more difficult. In the sample source file, direct embedded macros are avoided since they blossom into full assembled code during compilation. Every working register and its upper half (bits 16 through 31) is used. The data is brought into the lower sixteen bits by judicious use of rotate instructions whenever it is needed for further processing.

- Due to this register shortage, the sample implementation uses even BP in the AutoScan code implementation. Depending upon the core BIOS, it might be unacceptable to use the BP register or some other register that is used in this sample. In such an implementation, changes might be needed in the AutoScan code.

2.6 System BIOS (F-Segment) Shadowing Issues

System BIOS segment shadowing is almost always necessary to yield acceptable performance, because most systems use 8-bit BIOS chips and access to ROMs are traditionally slower than access to the DRAM subsystem. The partition of PCI bridge and memory controller into separate physical entities makes the shadowing algorithm in the 82450 PCIset more complex.

The following sequence of steps makes up the shadow algorithm in the reference BIOS implementation:

1. Configure the memory attribute register (MemoryAttrReg0) of the 82452 memory controller to enable memory writes to F-Segment. At this stage of programming, memory reads to the F-Segment go down to a buffered ISA bus, called the X bus, through the compatibility 82454 PCI bridge, and memory writes to F-Segment go to shadow DRAM at F-Segment.
2. Copy the data of F-Segment ROM into shadow RAM at F-Segment by executing a string copy. At this point, the BIOS must enable shadowing by turning off ROM accesses and enabling memory read accesses to shadow RAM at F-Segment. This task is difficult because the DRAM controller is a different physical entity with its own set of registers. By enabling DRAM accesses before turning off ROM accesses, ROM data coming through the compatibility bridge and shadow RAM data coming from the 82452 memory controller both drive the Pentium Pro processor bus for a brief period. This condition is not stable and can result in a system crash. So, instead of executing the shadow enabling portion of the code out of ROM, proceed along the following steps and execute this code out of DRAM.
3. Copy the code that turns off ROM Read accesses and enables shadow RAM into DRAM at 4000:0h. The BIOS has control of the DRAM resources at this point and the operating system is not loaded so this action is likely to be safe. However, problems with using the explicit address 4000:0h for this purpose could occur in some core BIOS architectures because of compression and such things. In these cases, any convenient DRAM address can be chosen so long as about 200h bytes are available for storage. This temporary storage is needed only until shadow is enabled.
4. Call this code in DRAM through a FAR call.
5. Control goes to a DRAM routine that turns of the ROM accesses and enables the ROM shadow. Then, control returns to ROM through a FAR return. In the DRAM routine, use fully embedded macros for accessing the 82450 PCIset registers because the other access code is in F-segment and does not support an inter-segment call.
6. The last step is to write protect the F-Segment DRAM.

3
Pentium® Pro Single Processor Initialization

This chapter describes initialization for systems with a single Pentium Pro processor as the CPU.

3.1 Cache Management and Memory Type Range Registers

The Pentium Pro processor has an internal L2 cache. The BIOS programmer must program the CPU's internal registers to set various memory attributes. Pentium Pro processor architecture also introduces two more newer memory attributes in addition to the usual Write Through (WT) and Write Back (WB) types. The new memory attributes that are added are Write Combining (WC) and Write Protected (WP). The WC type is used on frame buffers and the WP type is used for ROM shadow regions. Refer to the *Pentium Pro Processor Operating System Writer's Guide* for a full description of Memory Type Range Registers (MTRRs).

3.2 MTRR Management

Memory Type Range Registers are written and read in the same way as the conventional Machine Specific Registers (MSRs) of a typical Pentium processor by using RDMSR and WRMSR instructions. MTRRs are 64-bit registers and are divided into fixed and variable MTRRs. The MTRR capability register is examined to find out whether or not fixed MTRRs are implemented. Fixed MTRRs deal with memory between 0 and 1 megabyte, driven by the Setup menu, and they have been packaged in such a way as to implement PC-compatible address ranges easily. Variable MTRRs deal with memory above 1 megabyte.

In the reference implementation, a CMOS database that is Setup menu driven programs the memory types into fixed MTRRs. The code can also be used to configure 0- to 640-kilobyte base memory as WC type memory, for use in debugging the hardware efficiently. In a production BIOS, using WC is only an option for video RAM.

The reference BIOS uses variable MTRRs to cache memory above 1 megabyte. These registers can be used to specify a base address and a mask.

Although the user can change the memory type for a given address range of the memory by using the Setup menu, the reference BIOS implementation uses the default settings in Table 3-1.

Table 3-1. Default Memory Ranges for Memory Types

Memory Range	Use
0-640 kilobytes	configured as Write Back
B Segment (0B0000h-0BFFFFh)	WC
C,D Segments	WP if we do ROM shadow for these regions
E and F segments	WP
1 megabyte to top of memory	Write back
APIC address range	Uncached

The BIOS programmer must be careful while writing to MSRs because writing to undefined MTRRs, which are MSRs, generates an exception. For this reason, it is important to check whether variable and fixed MTRRs are implemented in a particular Pentium Pro processor machine and to check how many of these are implemented. All this information can be obtained by reading the MTRRcap register. Keep the following guidelines in mind when designing this code module:

- MTRR initialization is not lost upon assertion of INIT pin or on INIT IPI messages sent using the local APIC bus. MTRR registers go to their default values only upon assertion of a hard reset or at power-on.
- In a hardware environment, any other silicon that needs to be aware of the caching environment must be homogeneous with the MTRR based information. This uniformity of caching information becomes more important if a chipset other than the 82450 PCIset is used with the Pentium Pro processor CPU.
- The Pentium Pro processor local APIC address range must be set up as Uncached. The local APIC register access uses memory-mapped I/O which requires the range to be Uncached.
- Mask register calculation of a variable MTRR is complex. In the reference BIOS and implementation, the memory size is determined by reading the 82450 PCIset registers and then processing the value to get the physical mask that must be used. Refer to the sample code for more details.
- The Pentium Pro processor variable MTRR registers provide for specifying discontinuous memory ranges depending upon the mask provided. In general, the BIOS should ensure that discontinuous MTRRs are not enabled accidentally due to specifying an improper mask value for MTRRs. A shrink-wrapped operating system may refuse to boot or enable specific advanced features, such as mapping a linear frame buffer using the write-combining memory type (WC) on a platform that utilizes discontinuous MTRRs. This feature should only be used in special circumstances, such as for a 4KB uncacheable memory hole that occurs every 16MB throughout the entire 0-4GB address range. This type of memory hole does not normally occur on PC architecture platforms.
- It is acceptable to start from address 0 for a variable MTRR, even if any fixed MTRRs are already defined for this range, because the fixed MTRRs always override the variable ones.

Follow these guidelines while initializing the MTRRs of the Pentium Pro processor:

- A global enable bit, the FE bit in the MTRRdefType register, is used to enable all of the fixed range MTRRs. Each variable MTRR register pair (base and mask) is enabled by setting the V bit (bit 11) in the particular mask register. The enable bits (FE bit and the V bit) of MTRRs have default values indicating that they are disabled upon power-on reset; but the values of other bits in any MTRR are undefined. So, the BIOS code must initialize all the MTRRs to a known state by clearing all the fixed and variable MTRRs to zero before starting the initialization. This conservative method of initialization ensures that no MTRR is left in an undefined power-on state.
- BIOS code must clear the mask bits and base address bits of all variable MTRRs before using a specific variable MTRR. In addition, while changing a particular variable MTRR, the BIOS must be certain that its mask is disabled before attempting to change its base.
- The BIOS must reserve at least two variable MTRR register pairs (physbase and physmask) for operating system use. These must be the last two variable MTRRs. For example, if there are 8 variable MTRRS, 0-7, then MTRR pairs 6 and 7 are reserved for operating system use. If the BIOS does not reserve these, the operating system may not boot. But, initialization of these operating system reserved MTRRs to zero must be done by the BIOS as a safety measure.
- The BIOS should set MTRRdefType to UC. It is dangerous to set MTRRdefType as any other memory type due to the possibility of programming errors and speculative execution by the processor in areas where memory does not exist. The ability to set the default memory type to WB was designed for large memory systems with some holes. However, this function is now better served by the overlapping UC memory type capability.

3.3 Variable MTRR Initialization Algorithm

Fixed MTRRs control caching of memory below 1 megabyte and are set up according to Table 3-6. Variable MTRRs control the caching attributes of memory above 1 megabyte.

The basic architecture of MTRR base and mask registers is explained in the *Pentium Pro Processor Programmer's Reference Manual*. There are several methods of implementing this algorithm. The steps needed for the implementation used in the reference BIOS are:

1. Read MTRRCap register and get the count of variable MTRRs implemented in a particular CPU implementation. Initialize them to zero. The BIOS needs to reserve the top two variable MTRR pairs for operating system use. If all available MTRR pairs (range and mask) are consumed in implementing the BIOS algorithm itself, then some operating systems will not boot.
2. Find the total amount of memory by reading the memory controller registers. This assumes that the AutoScan module has been run and all memory registers that control the size of the system memory have been initialized. We can assume this since the AutoScan module runs very early in POST with caches disabled. In a Pentium Pro processor and 82450 PCIset based implementation, total installed memory can be found by reading the DramRowLimit7 register.
3. Set the first MTRRBaseRegister to 0 and figure the largest power of two portion of memory that can be carved out of the total memory installed. For example, if the total memory is 384 megabytes, then the largest power of two portion that can be carved out is 256 megabytes.

4. Figure out the mask for this largest power of two portion of memory. This is done in the SetVarMTRRMaskReg procedure in the reference BIOS, which sets the mask to satisfy the address match comparison rule that may be described as follows:

   Address AND MTRRphysMask = (MTRRphysbase AND MTRRphysMask).

   The mask is set by the routine such that all addresses in this carved power of two portion of memory satisfy this equation and will be cached according to the memory type set in the lower portion of MTRRphysbase register.
5. Compute the remaining amount of memory and carve out another largest possible power of two portion. Repeat Step 4 to find the mask for this portion after setting the next available MTRRphysbase register to the top of the address that was cached in the previous step.
6. Repeat Steps 4 and 5 until all the installed cacheable memory in the system is exhausted or you run out of MTRR register pairs. Remember that in a Pentium Pro processor implementation that has eight MTRR pairs, only six of them are available for BIOS use.

Here is an example with numerical values. In a system with 96 megabytes of total memory, the MTRRPhysBase/Mask registers have the following values after executing the MTRR initialization algorithm:

- MTRRPhysBase0 = 0000_0000_0000_0006h
- MTRRPhysMask0 = 0000_000F_FC00_0800h

  This caches the first 64 megabytes of memory as WB memory type.
- MTRRPhysBase1 = 0000_0000_0400_0006h
- MTRRPhysMask1 = 0000_000F_FE00_0800h

  This caches the next 32 megabytes of memory as WB memory type.

3.4 Local APIC Initialization

The Pentium Pro processor contains an integrated local APIC device in order to support a multiprocessor environment. This local APIC receives interrupt-related messages. Intel recommends that the local APIC be initialized to virtual-wire mode, as required for compliance with the *MultiProcessor Specification*, even in a single processor environment.

Noteworthy issues in APIC initialization are:

- All local APIC registers are reset to default values upon any hard CPU reset. INIT signal assertion and INIT IPI messages reset all local APIC registers except APIC BASE and APIC ID registers. Local APIC initialization code must be hooked to core BIOS routines such that a local APIC gets initialized to virtual-wire mode every time the CPU is reset (both hard and soft resets).
- APIC address space is relocatable, unlike the APIC registers of the Pentium 100 processor. It is possible to remap Pentium Pro processor's APIC addresses to space below 1 megabyte and program the APIC in real mode. Code developed to be portable across a group of related processors should continue to program the APIC in protected mode and not relocate the APIC base address.
- Set the memory type in MTRR for the address range where APIC registers reside as the Uncached (UC) type.

3.5 Pentium Pro Machine Check Architecture

The Pentium Pro processor has enable and status bits for several machine check errors that could occur during the operation of the processor. Potentially, this feature allows the processor to execute a graceful shutdown. In most cases, the machine check provides information about errors that occurred during the processor's normal operation. Generally, a machine check exception handler reads the enable and status bits of the architecture, logs the appropriate data, and provides user-based error messages. The *Pentium Pro Processor Operating System Writer's Guide* describes the use of machine check architecture in detail.

The MC0_CTL register contains bits that can be set by the BIOS since their functions are platform dependent and the BIOS knows the platform architecture. An operating system should not set bits in the MC0_CTL register . An operating system may reset (clear) a bit in MC0_CTL to disable reporting of specific errors.

3.6 Pentium Pro Processor Common Setup Information

Information in the Setup menu usually is specific to a particular BIOS implementation. CMOS resource allocation is done bit-wise in most modern BIOS architectures to prevent a waste of precious CMOS space and to allow more Setup items to be added.

Following is a list of new Setup menu items that support Pentium Pro processor and 82450 PCIset features.

- Number of processors as found by the MP algorithm, with an option to take some off the MP table and hence off the booting process.
- Write Protect memory types (enable/disable) for all ROM spaces.
- WC memory types (enable/disable) for frame buffers.
- Definition of cacheable/uncacheable ranges for use with physical MTRRs.
- Deturbo timer in the 82450 PCIset.
- PCI features of the 82454 PCI bridge, such as write posting for inbound and outbound PCI transactions.
- Several Error Reporting features of the 82454 PCI bridge.
- Memory Error Correction features of the 82452 memory controller.
- Memory Bandwidth enhancement features of the 82452 memory controller, like page open policy.
- SM RAM ranges.

It is not necessary to give a setup choice for every 82450 PCIset feature that is enabled. The setup feature may be nice in an evaluation BIOS but may be less desirable in a production BIOS. The reference BIOS implementation gives a setup choice for every 82450 PCIset feature, since it was designed to be a part of evaluation BIOS.

4
Pentium Pro Multiprocessor Initialization

These features of the BIOS multiprocessor (MP) initialization algorithm are described in this chapter:

- It does not depend on the number of processors installed in the system, but is capable of figuring out the number of processors installed in the system. The algorithm described works for a maximum of fifteen processors.
- It has the ability, through the Setup menu, of taking a requested number of processors off the system without actually unplugging them from their sockets.
- It can do limited testing on each of the processors and initializes their MTRRs.
- It makes sure that the MTRRs of all the processors in an MP system have identical values by using a common CMOS database.
- It loads processor specific BIOS update data to all the processors installed in the system. See Chapter 8 for more details on the BIOS update feature of Pentium Pro processor CPUs.
- It builds the MP table for use by the operating system, as described in the *MultiProcessor Specification*.
- This algorithm needs RAM for its operation. This limitation is not imposed by the algorithm, since RAM is required for building the MP table anyway. The RAM for building the MP table can be chosen either in the extended BIOS data area or in the F-Segment shadow RAM. The size of the RAM table depends on the system configuration.

4.1 Multiprocessor Initialization in BIOS

The Pentium Pro processor is MP-ready, so proper MP initialization is an essential part of the Pentium Pro processor BIOS. The Pentium Pro processor implements an MP initialization protocol during which processors communicate with each other, elect a single processor to be a boot processor, and put the auxiliary system processors in a loop waiting for StartUp Inter Processor Interrupts (IPIs). This wait state, described in the *Pentium Pro Operating System Writer's Guide*, is a special state wherein the processor is not executing, but its local APIC is listening to StartUp IPI messages on the APIC bus. This MP Boot Protocol is executed only on a processor reset signal toggle. It is not re-executed upon receipt of Init APIC messages. In other words, the boot processor is not renegotiated on receipt of an INIT IPI message.

4.2 MP Initialization Algorithm

This section describes the algorithm that is used by the reference BIOS for initializing Pentium Pro processor-based multiprocessor systems.

Upon reset, the Pentium Pro processors elect a BSP from among themselves and put the other processors into a wait state. Processors in the wait state wait for StartUp inter-processor interrupts on their local APIC bus.

4.2.1 Algorithm for Bootstrap Processor

The bootstrap processor (BSP) performs these parts of the algorithm:

1. The elected BSP starts executing the BIOS, proceeds through POST and other BIOS components, and enters the MP algorithm code.
2. The BSP initializes a predefined RAM location to a value of 1. In the reference BIOS, this location is called the CPUCounter. The BSP initializes the LockSemaphore routine to Vacant = `00h` for use by a module that initializes all non-BSPs.
3. The BSP sends a StartUp APIC message broadcast to all other processors by programming the interrupt command register with a vector that points to the FindAndInitAllCPUs module.
4. The BSP waits for the auxiliary processors to complete their initialization (described in detail in the next section). This wait loop can be implemented in various ways. It can be a simple software wait loop that is large enough to allow auxiliary processors to complete their initialization. It can use the BIOS timer (a hardware timer for use by the BIOS that is implemented by all Intel chipsets) to implement an exact delay time loop. The local APIC of the BSP also contains a timer that can be used for wait loop implementation.

   Finally, a more complex wait algorithm can be implemented as follows.

   a. The auxiliary processors increment a CPUCounter (a memory location initialized to 01h by the BSP at the start of the algorithm) as soon as they start their initialization loop, as described in the next section. The BSP examines this location using a synchronized lock read every two seconds (two second poll frequency is a suggested value and individual BIOS implementations can tune them according to their requirements). The BSP compares this CPU counter value on a particular read to the value of the same variable during the previous read. If it has not changed, all processors are finished with their initialization. This BSP wait loop algorithm assumes an auxiliary processor takes no longer than two seconds to complete initialization. The CPUCounter value at the end of the algorithm reflects the total number of CPUs in the system, including the BSP.

   b. When the BSP exits the timing loop, the CPUCounter contains the number of processors in the system, and all the processor entries in the MP table are built. Refer to the description of FindAndInitAllCPUs for details. At this stage, all auxiliary processors are initialized with an identical set of MTRRs and other CPU registers.

   c. The BSP reads the CMOS RAM initialized through setup to determine the number of processors that the user wants in this boot session. The BSP then constructs the remainder of the MP table and removes or disables the MP table entries for the non-operational processors. For example, if the user wants only two processors out of four populated ones, MP entries corresponding to two processors are removed or disabled. In our algorithm, an INIT IPI is sent to all the auxiliary processors, causing them to wait for a STARTUP IPI event. INIT IPI must be sent to auxiliary processors after System Management Mode (SMM) initialization of auxiliary processors is completed (if SMM is enabled in CMOS based setup). The SMM initialization procedure is detailed in Chapter 5. The operating system consults the MP table entries to find enabled processors and sends them a STARTUP IPI.

   d. The MP table checksum is calculated for the adjusted MP table. Other parameters, such as the number of entries and the length of the MP table, are also determined.

4.2.2 Algorithm for Auxiliary Processor Initialization

The algorithm for auxiliary processor initialization is implemented in a module called FindAndInitAllCPUs in the reference BIOS and has the following responsibilities:

- Although all CPUs enter this algorithm almost simultaneously, this module uses Synchronization Locks to let processors one-by-one into the rest of the algorithm.
- Each CPU runs CPUID instruction and initializes its MTRRs by reading the CMOS data base.
- Each CPU constructs its particular entry in the MP table. This entry includes information on the CPUID, and the address and version of the local APIC.
- Each CPU increments the CPUCounter (a memory variable initialized by BSP), so that the BSP knows the number of processors in the system.
- If System Management Mode (SMM) is enabled, it is initialized. Refer Chapter 5 for more details.

The algorithm performs these actions:

1. All auxiliary CPUs wake up simultaneously after listening to the StartUp IPI broadcast from the BSP and start executing this module.
2. The first processor that executes the TestLock procedure sets the LockSemaphore routine to NotVacant=`0ffh` and proceeds to Step 3. All other processors wait in the TestLock procedure until the processor that moved to Step 3 completes and unlocks the lock semaphore variable. The TestLock procedure can be coded as:

```
                Mov al, NotVacant
TestLock:
                Xchg Byte ptr [LockSemaphore], al
                Cmp al, NotVacant
                Jz TestLock
                ----To Step (3)
```

NOTE. *On Intel Architecture processors, the* `Xchg` *instruction has a built-in lock feature.*

3. The first auxiliary CPU to emerge from the TestLock procedure increments the CPUCounter variable.
4. The auxiliary CPU is channeled through the same routines that initialized the BSP to initialize the MTRRs from the common CMOS data base.
5. The auxiliary CPU runs CPUID to know its features.
6. The auxiliary CPU constructs its entry in the MP table based on its CPUID, Local APIC ID, and so on.
7. The auxiliary CPU loads any BIOS update data to the processor. See Chapter 8 for more details on the BIOS processor update feature of Pentium Pro processor CPU.
8. The auxiliary CPU releases the lock on the semaphore by executing the ReleaseLock routine. This action allows the next waiting CPU into the initialization loop:

```
ReleaseLock:
                Mov al, Vacant
                Xchg Byte ptr [LockSemaphore], al
                ---To Step (9)
```

9. The auxiliary CPU goes into a wait loop. If SMM of the auxiliary processor is enabled in the CMOS based setup, the CPU receives an SMI message from the BSP when the BIOS performs SMM initialization (as explained in Chapter 5). If SMM of the auxiliary processor is disabled, BSP sends an INIT IPI message to the auxiliary processor, which puts the initialized CPU into a special state wherein it halts and waits for a STARTUP IPI message from the operating system.

5
Pentium Pro Processor System Management Mode Initialization

The Pentium Pro processor implements System Management Mode (SMM), which helps system developers provide very high level systems functions, such as power management or security, in a manner that is transparent not only to the application software but also to the operating systems.

This chapter describes the initialization of SMM for both single and multiprocessor systems.

5.1 Single Pentium Pro Processor SMM

The Pentium Pro processor CPU can accept both synchronous and asynchronous system management interrupts (SMI) that switch the processor to SMM. A synchronous SMI usually is generated because of an I/O trap by the chipset or additional hardware. The chipset must provide a register with status bits to indicate the occurrence of a synchronous SMI. The 82450 PCIset currently does not support synchronous SMIs. SMM initialization is done at a later part of the POST.

The 82450 PCIset does not provide any processor address translation during SMM. The registers inside the 82452 memory controller need to know the SMM address range in order to protect the address range from any other processor modes.

During initialization, the operating system is not loaded and the RAM resource is owned by the BIOS. When the BSP executes the SMM initialization module, all of the chipset initialization is complete and RAM is available.

The following steps explain an algorithm for initializing the SMM of a single processor system.

1. The CPU starts up the InitSMM software module. This module is responsible for the SMM initialization. First, the InitSMM module copies a small section of code, the SMBaseInit module, to the Default SMI Vector at `3000h:8000h`. At this point, the operating system is not booted and all the memory is owned by the BIOS, so this operation does not destroy any operating system or program data. The CPU also initializes a flag, indicating that SMM initialization is not yet finished.
2. In order to set up the SMM base, the CPU sends itself an SMI through its local APIC by programming the interrupt command register, and then starts waiting on a flag that indicates that the CPU SMM is not initialized. Rather than waiting on the flag eternally, use a watch dog timer that makes use of the local APIC's timer resource. If this timer times out and the flag indicates that SMM is not initialized, then SMM initialization has failed.
3. The APIC-based SMI reaches the BSP core, which enters SMM and starts executing at `3000h:8000h`. In other words, the SMBASEInit module has been started by the BSP in SMM. The 82450 PCIset need not be programmed to enable SMI, because no support is required from the 82450 PCIset at this point.

4. The SMBASEInit module initializes the SMBASE slot in the SMM dump area, at address cs:0FEF8h, to the required SMI vector address. Ideally, this address is read from the Setup menu, to be sensitive to user's configuration needs. For this example, assume that it is 0A000h.
5. Copy the SMMHandler module to the SMM address range. In this example, this address is 0A000:8000h. Program the memory attribute register at the 82452 controller CSE Offset 58h to enable the memory range. This SMMHandler contains the code that handles an SMI, and it is implementation-specific, depending on the purpose for which the SMI is used in a given system.
6. Turn off the enable bits of the Memory Attribute register at 82452 memory controller CSE Offset 58h, since we enable it as SM RAM in the 82452 controller later on.
7. The SMBASEInit module updates a flag to indicate BSP SMM is initialized and then executes an RSM instruction.
8. Control returns to the InitSMM module, which exits the waiting loop in the InitSMMmodule because of the status of the initialization flag.
9. The algorithm programs the 82450 PCIset's 82452 memory controller registers to give them information about the SMM address range by programming the SMRAM Range Register at 82452 memory controller CSE Offset 0B8h. Then, enable the SMM ability by programming the 82452 memory controller register at CSE offset of 57h.

5.2 Pentium Pro Multiprocessor SMM Initialization

The algorithm presented here is capable of initializing the SMM of a Pentium Pro processor-based multiprocessor system. It requires a 32-kilobyte SMM slot per processor. For example, if a system contains four Pentium Pro processors and a decision has been made to use A and B segments of the shadow memory for SMM, then the SMBASE registers of the four processors can be successively programmed to 98000h, 0A0000h, 0A8000h and 0B0000h. Note that, although the SMBASE of the first processor is initialized to 98000h, it still uses memory within A and B segments, since SMM entry points are at SMBASE+8000h (SMBASE+EIP wherein EIP=8000h on entry to SMM mode). As more processors are added, the demand for SM RAM also goes up and it may become necessary to locate SMM code just below the top of installed memory.

The SMM initialization module runs after MP initialization and has access to the MP table constructed during MP initialization. In other words, the BIOS has all the information about the number of processors and their respective APIC Ids. Note that SMI APIC messages are accepted by the auxiliary processors only when they are executing code. The MP initialization module, described in Chapter 4, puts the auxiliary processors into a wait loop (if SMM is enabled in the Setup menu) and defers sending INIT IPI messages to auxiliary CPUs until the SMM initialization is completed.

The BSP's SMM initialization is done exactly as explained in the previous section on single processor initialization. After the BSP's SMM initialization is complete, the remainder of the MP algorithm follows these steps:

1. Once BSP's SMM is initialized, the BSP consults the MP table that is already constructed by the MP initialization module. The BSP gets information about the number of processors to be initialized and about their APIC IDs from this consultation.
2. To indicate the number of the processors being initialized, the BSP sets a flag that is readable by SMMBaseInit module.

3. The BSP sends an SMI APIC message to the next processor to be initialized by using its APIC ID and programming the interrupt command register and starts waiting on a flag, indicating that the next processor SMM is not initialized. Instead of waiting on the flag eternally, use a watch dog timer that makes use of the local APIC's timer resource. If this timer times out and the flag indicates that SMM is not initialized, the SMM initialization for this processor has failed.
4. The next processor enters SMMBaseInit module. The module consults its flags to figure out that this is incremental initialization of the next processor. The SMMBaseInit module initializes the SMBASE slot in the SMM core dump at address `cs:0FEF8h` to the PreviousSMBaseAddress + 32 kilobytes.
5. The SMMBaseInit module copies the SMM Handler code to Previous SMM HandlerAddress + 32 kilobytes.
6. The SMMBaseInit module updates flags for the use of the BSP and executes an RSM instruction, returning to its wait loop.
7. Once again, the MP table is consulted to find information about the next processor and the initialization process is repeated.
8. Continue this procedure until the MP table indicates that no more processors are to be initialized.
9. Send an INIT IPI message broadcast to all auxiliary processors.

5.3 Pentium Pro Multiprocessor SMM Handler

The SMM handler is the code module that services the system management interrupt (SMI).

At least two types of implementations can be used in a typical Pentium Pro multiprocessor SMM handler:

- Only the BSP executes SMM.
- Any processor can execute SMM.

5.3.1 Only the BSP Executes SMM

The following steps explain the structure of an SMM handler when only the BSP is allowed to execute the SMM handler:

1. When an SMI occurs in the system, all processors enter their respective SMM spaces. All processors but the BSP are held waiting on flags in their SMM data area.
2. The BSP continues execution of the SMM handler and carries it to completion.
3. The BSP resets and releases the flags on other processors.
4. All processors execute an Resume (RSM) instruction and return to the state that they were in before the SMI interruption.

5.3.2 Any Processor Can Execute SMM

The following steps explain the structure of an SMM handler, where any one of the processors is allowed to execute the SMM handler.

1. When an SMI occurs in the system, all processors enter their respective SMM spaces. They race to a TestLock procedure. This procedure is often described as "race to a flag" algorithm and is explained in Chapter 4. Only one processor gets to a TestLock procedure first and starts executing the SMM handler. We call this processor the SMM processor.
2. All other processors are held off in a waiting loop, waiting on a flag until the SMM processor finishes the SMM code execution and releases the lock.
3. The SMM processor continues execution of the SMM handler and carries the SM handler to completion.

The SMM processor resets and releases the flags (locks) on other processors. The SMM processor also sets an SMMDone flag. Other processors released from the loop examine the SMMDone flag and execute an RSM instruction. The BSP also exits SMM by executing an RSM instruction. Whatever method of implementation is chosen, all processors are brought into SMM simultaneously.

If some processors are left in normal mode and others are brought into SMM, the operating system will crash if it is not designed to lose some of its processor resources involuntarily. This is true in most of the current operating system designs.

6
Large Memory Support

This chapter describes the call interface for memory larger than 64 megabytes.

6.1 Description of the Interface

If an ISA bus system has more than 64 megabytes of memory, there is no clean way of reporting it to the operating system. EISA systems have special INT 15 calls that do this. Intel and Microsoft* developed a new INT 15 call mechanism (AX=0E820h) to overcome this problem and supply the operating system with a clean memory map. This memory map indicates address ranges that are reserved and ranges that are available in the motherboard. This call is only available in real mode and currently is used by the Windows NT* operating system to get memory size and reserved address ranges. Appendix A explains the specifications of the call.

6.2 Implementation Issues

Since the call is a simple real mode call and not a dual 16/32-bit mode call, be aware of the following:

- The call is chipset dependent since the BIOS must report any memory holes created by the programming memory gap registers of a typical chipset.
- In order to implement this call, the core BIOS must support the full memory size in some CMOS location. The BIOS needs full memory size information for use in EISA memory reporting calls.

7
Register Editor Program

The Pentium Pro processor Helper register edit program (`P6H.EXE`) is a DOS-based TSR. It is a very powerful tool for debugging and tuning system performance because it enables the user to change the Pentium Pro processor-based platform registers.

The main features of the register editor are:

- The Read/Write commands of the editor take wild cards, such as `*` and `?` .
- The editor enables the user to create a file containing the values of all the registers in the system.
- The editor program operates on a compressed register table that is built by a small Table Creator program called `p6htab.exe`. This program reads in an editable text file called `p6h.reg`. Therefore, the end user can add new registers without modifying the source code.

NOTE. *To boot any operating system besides MS-DOS*, edit the register bits under MS-DOS and use the INT 19 instructions assembled under MS-DOS debug.*

8
Pentium Pro Processor BIOS Update Feature

The Pentium Pro processor has the capability to correct specific errata through the loading of an Intel-supplied data block. This chapter describes the underlying mechanisms the BIOS needs in order to utilize this feature during system initialization. It also describes a specification that provides for incorporating future releases of the update into a system BIOS.

Intel considers the combination of a particular silicon revision and BIOS update as the equivalent stepping of the processor. Intel completes a full-stepping level validation and testing for new releases of BIOS updates.

An update loader integrated within the BIOS uses data from the update to correct specific errata. The BIOS is responsible for loading the update on all processors during system initialization, as shown in Figure 8-1.

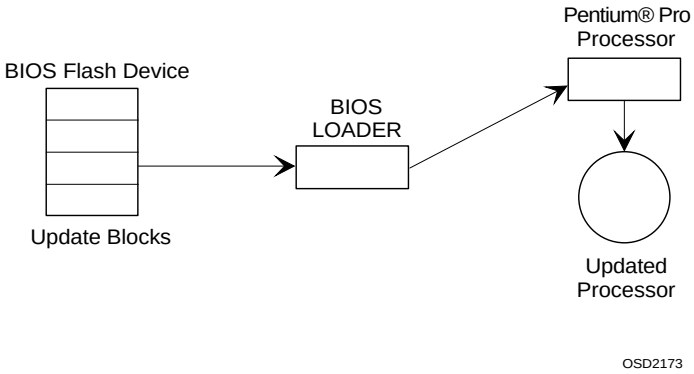

Figure 8-1. Integrating Processor Specific Updates

8.1 BIOS Update

A BIOS Update consists of an Intel-supplied binary that contains a descriptive header and data. No executable code resides within the update. This section describes the update and the structure of its data format.

Each BIOS Update is tailored for a particular stepping of the Pentium Pro processor. The data within the update is encrypted by Intel and is designed such that it is rejected by any stepping of the processor other than its intended recipient. Thus, a given BIOS Update is associated with a particular family, model, and stepping of the processor as returned by the CPUID instruction. The encryption scheme also guards against tampering of the update data and provides a means for determining the authenticity of any given BIOS update.

The BIOS Update is a data block that is exactly 2048 bytes in length. The initial 48 bytes of the update contain a header with information used to maintain the update. The update header and its reserved fields are interpreted by software based upon the Header Version. The initial version of the header is `00000001h`. Figure 8-2 shows the format of the BIOS Update data block, and Table 8-1 explains each of the fields.

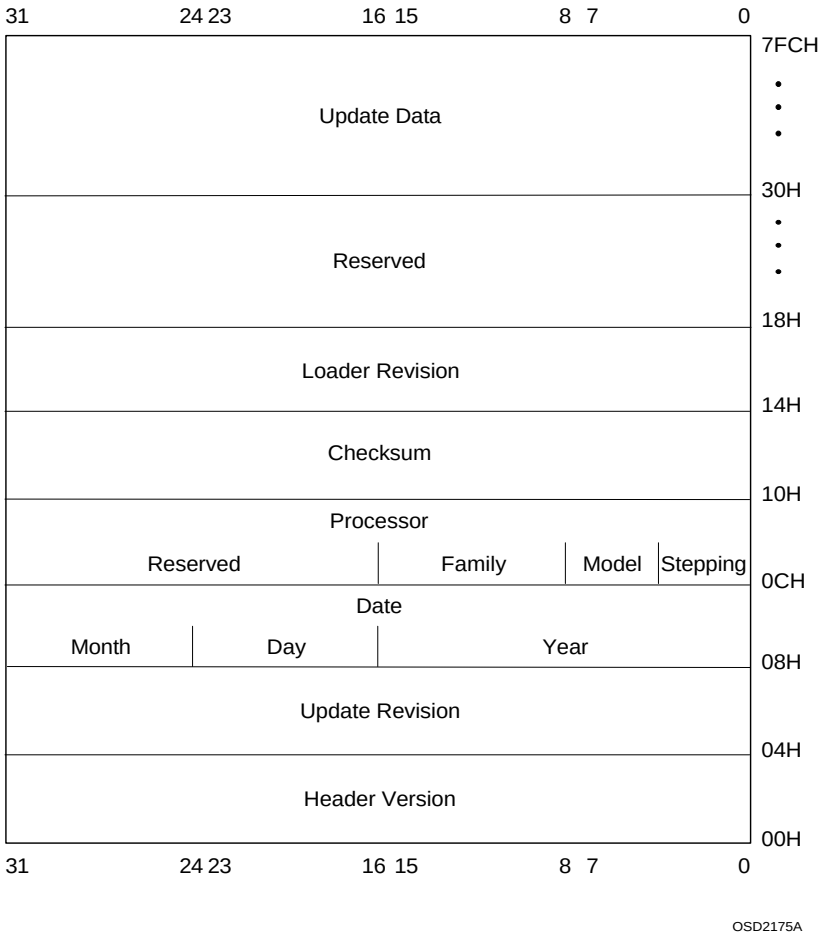

Figure 8-2. BIOS Update Data Block

Table 8-1. BIOS Update Header Data

Field Name	Offset (in bytes)	Length (in bytes)	Description
Header Version	0	4	Version number of the update header.
Update Revision	4	4	Unique version number for the update, the basis for the update signature provided by the processor to indicate the current update functioning within the processor. Used by the BIOS to authenticate the update and verify that it is loaded successfully by the processor. The value in this field cannot be used for processor stepping identification alone.
Date	8	4	Date of the update creation in binary format: mmddyyyy, month, day year (e.g., 07/18/95 as 07181995h).
Processor	12	4	Family, model, and stepping of processor that requires this particular update revision (e.g., 00000611h). Each BIOS update is designed specifically for a given family, model, and stepping of the processor. The BIOS uses the Processor field in conjunction with the CPUID instruction to determine whether or not an update is appropriate to load on a processor. The information encoded within this field exactly corresponds to the bit representations returned by the CPUID instruction.
Checksum	16	4	Checksum of Update Data and Header. Used to verify the integrity of the update header and data. Checksum is correct when summation of the 512 double words of the update results in the value zero.
Loader Revision	20	4	Version number of the loader program needed to correctly load this update. The initial version is 00000001h.
Reserved	24	24	Reserved Fields for future expansion.
Update Data	48	2000	Update data.

8.2 Update Loader

This section describes the update loader used to load an update into the Pentium Pro processor. It also discusses the requirements placed upon the BIOS to ensure proper loading of an update.

The update loader contains the minimal instructions needed to load an update. The specific instruction sequence required to load an update is associated with the Loader Revision field contained within the update header. The revision of the update loader is expected to change very infrequently, potentially only when new processor models are introduced.

The code below represents the update loader with a Loader Revision of 00000001h:

```
mov      ecx,79h            ; 79H in ECX
xor      eax,eax
xor      ebx,ebx
mov      ax,cs              ; Segment of BIOS Update
shl      eax,4
mov      bx,offset Update   ; Offset of BIOS Update
add      eax,ebx            ; Linear Address of Update in EAX
add      eax,48d            ; Offset of the Update Data within the Update
xor      edx,edx            ; Zero in EDX
WRMSR                       ; BIOS Update Trigger
```

8.2.1 Update Loading Procedure

The simple loader previously described assumes that Update is the address of a BIOS update (header and data) embedded within the code segment of the BIOS. It also assumes that the processor is operating in real mode. The data may reside anywhere in memory that is accessible by the processor within its current operating mode (real, protected).

Before the BIOS executes the Update trigger (WRMSR) instruction the following must be true:

- EAX contains the linear address of the start of the Update Data
- EDX contains zero
- ECX contains 79h

Other requirements to keep in mind are:

- If the update is loaded while the processor is in real mode, then the update data may not cross a segment boundary.
- If the update is loaded while the processor is in real mode, then the update data may not exceed a segment limit.
- If paging is enabled, then the update data must be mapped by pages that are currently present.
- The BIOS update data does not require any particular byte or word boundary alignment.

8.2.1.1 Hard Resets in Update Loading

The effects of a loaded update are cleared from the processor upon a hard reset. Therefore, each time a hard reset is asserted during the BIOS POST, the BIOS update must be reloaded on all processors that observed the reset. The effects of a loaded update are persistent across a processor INIT. No ill effects are caused by loading an update into a processor multiple times.

8.2.1.2 Timing of Update Loading

There are no specific timing requirements as to when the BIOS must load the update into the processor. It is not necessary to load the update at the time of initial power-on reset or very early during the POST stage. Intel guarantees that the processor successfully executes through POST without having the update loaded. Intel recommends that BIOS vendors provide a setup option to enable/disable update loading. This recommendation implies that update loading must occur after setup.

8.2.1.3 Update in a Multiprocessor System

A multiprocessor (MP) system requires loading each processor with update data appropriate for its CPUID. The BIOS is responsible for ensuring that this requirement is met, and that the loader is located in a module that is executed by all processors in the system. If a system design permits multiple steppings of Pentium Pro processors to exist concurrently, then the BIOS must verify each individual CPUID against the update header information to ensure appropriate loading. Given these considerations, it is most practical to load the update during MP initialization.

8.2.1.4 Sample Loader

The actual loader implementation in a BIOS contains the basic loader presented in this section and additional supporting code. A sample implementation is supplied in the Pentium Pro processor reference source code kit. The sample loader in the source code kit verifies the CPUID values of various processors in the system and loads a correct update for each processor stepping that is found in the system.

8.2.2 Update Loader Enhancements

The update loader presented in the previous section is a minimal implementation that can be enhanced to provide additional functionality and features. Some potential enhancements are described below:

- The BIOS can incorporate multiple updates to support multiple steppings of the Pentium Pro processor. This feature provides for operating in a mixed stepping environment on an MP system and enables a user to upgrade to a later version of the processor. In this case, modify the loader to check the CPUID of the processor that it is running on against the available headers before loading a particular Update.
- A loader can load the update and test the processor to determine if the update was loaded correctly. This can be done as described in the Update Signature and Verification section in this chapter.
- A loader can verify the integrity of the update data by performing a checksum on the double words of the update summing to zero, and can reject the update.
- A loader can provide sign on messages indicating successful loading of an update.

8.3 Update Signature and Verification

The Pentium Pro processor provides capabilities to verify the authenticity of a particular update and to identify the Update Revision of the currently functioning revision. This section describes these model-specific extensions of the processor that support this feature. The update verification method below assumes that the BIOS only verifies an update that is more recent than the revision currently loaded into the processor.

The CPUID instruction has been enhanced to return a value in a model specific register in addition to its usual register return values. The semantics of the CPUID instruction are not modified except to cause it to deposit an update ID value in the 64-bit model-specific register (MSR) at address `08Bh`. If no update is present in the processor, the value in the MSR remains unmodified. Normally a zero value is preloaded into the MSR before executing the CPUID instruction. If the MSR still contains zero after executing CPUID, this indicates that no update is present.

The Update ID value returned in the EDX register after a RDMSR instruction indicates the revision of the update loaded in the processor. This value, in combination with the normal CPUID value returned in the EAX register, uniquely identifies a particular update. The signature ID can be directly compared with the Update Revision field in the BIOS Update header for verification of the correct BIOS update load. No consecutive updates released for a given stepping of the Pentium Pro processor may share the same signature. Updates for different steppings are differentiated by the CPUID value.

8.3.1 Determining the Signature

An update that is successfully loaded into the processor provides a signature that matches the Update Revision of the currently functioning revision. This signature is available any time after the actual update has been loaded, and requesting this signature does not have any negative impact upon any currently loaded update. The procedure for determining this signature is:

```
1. mov ecx, 08Bh      ;Model Specific Register to Read
2. xor eax,eax
3. xor edx,edx
4. WRMSR              ;Load 0 to MSR at 8Bh
5. mov eax,1
6. CPUID
7. mov ecx, 08BH      ;Model Specific Register to Read
8. RDMSR              ;Read Model Specific Register
```

If there is an Update currently active in the processor, its update revision is returned in the EDX register after the RDMSR instruction has completed.

8.3.2 Authenticating the Update

An update may be authenticated by the BIOS using the signature primitive, described above, with the following algorithm:

```
Z = Obtain Update Revision from the Update Header to be authenticated;
X = Obtain Current Update Signature from MSR 8Bh;
If (Z > X) Then
                Load Update that is to be authenticated;
                Y = Obtain New Signature from MSR 8Bh;
                If (Z == Y) then Success
                Else Fail
Else Fail
```

The algorithm requires that the BIOS only authenticate updates that contain a numerically larger revision than the currently loaded revision, where Current Signature (X) < New Update Revision (Z). This authentication procedure relies upon the decryption provided by the processor to verify an update from a potentially hostile source. As an example, this mechanism in conjunction with other safeguards provides security for dynamically incorporating field updates into the BIOS.

8.4 Pentium Pro Processor BIOS Update Specifications

This section describes two interfaces that an application can use to dynamically integrate processor-specific updates into the system BIOS. In this discussion, the application is referred to as the *calling program* or *caller*.

Both the real mode INT 15h call and the alternate protected mode call specifications described here are Intel extensions to an OEM BIOS. These extensions allow an application to read and modify the contents of the BIOS update data in NVRAM. The BIOS update loader, which is part of the system BIOS, cannot be updated by either of the two interfaces. All of the functions defined in either of the two specifications must be implemented for a system to be considered compliant with the specification. The INT15 functions are accessible only from real mode. The protected mode specification provides an interface that is available in either real or protected mode.

8.4.1 Responsibilities of the BIOS

If a BIOS passes the presence test (INT 15h, AX=0D042h, BL=0h) it must implement all of the subfunctions defined in the INT 15h, AX= 0D042h specification. If a BIOS implements the protected mode interface signature and associated parameters, it must implement all of the subfunctions defined in the protected mode interface specification. There are no optional functions. The BIOS must load the appropriate update for each processor during system initialization.

A Header Version of an update block containing the value `0FFFFFFFFh` indicates that the update block is unused and available for storing a new update.

The BIOS is responsible for providing a 2048 byte region of non-volatile storage (NVRAM) for each potential processor stepping within a system. This storage unit is referred to as an *update block*. The BIOS for a single processor system need only provide one update block to store the BIOS update data. The BIOS for a multiple processor capable system needs to provide one update block for each unique processor stepping supported by the OEM's system. The BIOS is responsible for managing the NVRAM update blocks. This includes garbage collection, such as removing update blocks that exist in NVRAM for which a corresponding processor does not exist in the system. This specification only provides the mechanism for ensuring security, the uniqueness of an entry, and that stale entries are not loaded. The actual update block management is implementation specific on a per-BIOS basis. As an example, the BIOS may use update blocks sequentially in ascending order with CPU signatures sorted versus the first available block. In addition, garbage collection may be implemented as a setup option to clear all NVRAM slots or as BIOS code that searches and eliminates unused entries during boot.

The following algorithm describes the steps performed during BIOS initialization used to load the updates into the processor(s). It assumes that the BIOS ensures that no update contained within NVRAM has a header version or loader version that does not match one currently supported by the BIOS and that the update block contains a correct checksum. It also assumes that the BIOS ensures that at most one update exists for each processor stepping and that older update revisions are not allowed to overwrite more recent ones. These requirements are checked by the BIOS during the execution of the write update function of this interface.

1. The BIOS sequentially scans through all of the update blocks in NVRAM starting with index 0. The BIOS scans until it finds an update where the processor field in the header matches the family, model, and stepping of the current processor.

2. For each processor in the system
   {
   Determine Family, Model and Stepping via CPUID instruction for Genuine Intel processors;
   for (i=Update Block 0, i< Update Num; i++) {
   If (UpdateHeader.Processor == Family, Model and Stepping of this processor) {
   Load the Update into the Processor from UpdateHeader.UpdateData;
   /* optionally verify that update was correctly loaded into the processor */
   /* Go on to next processor */
   Break;
   }
   /* If no match is found, then the current processor does not require an update OR */
   /* an update for that processor has not been loaded into NVRAM */
   }

When performing either the INT 15h, 0D042h functions or protected mode interface functions, the BIOS must assume that the caller has no knowledge about platform specific requirements. It is the responsibility of the BIOS calls to manage all chipset and platform specific prerequisites for managing the NVRAM device. When writing the update data via the Write Update subfunction, the BIOS must maintain implementation specific data requirements, such as the update of NVRAM checksum. The BIOS should also attempt to verify the success of write operations on the storage device used to record the update.

The BIOS must implement a mechanism whereby the update loading at initialization time can be disabled without destroying the update. Possible ways of implementing this function are either through CMOS bits or through bits located in the NVRAM device. Regardless of which mechanism allows for the enabling and disabling, it is recommended that the user be provided a method for determining the state of the update, such as via the BIOS setup routine.

8.4.2 Responsibilities of the Calling Program

This section of the document lists the responsibilities of the calling program using either of the two interface specifications to load BIOS update(s) into BIOS NVRAM.

The calling program must call the INT 15h, 0D042h functions from a pure real mode program and must be executing on a system that is running in pure real mode. The caller may call the protected mode interface from either pure real mode or protected mode. The caller must issue the Presence Test function (sub function 0) and verify the signature and return codes of that function. It is important that the calling program provides the required scratch RAM buffers for the BIOS and the proper stack size as specified in the interface definition.

The calling program must read any update data that already exists in the BIOS in order to make decisions about the appropriateness of loading the update. The BIOS refuses to overwrite a newer update with an older version. The Update Header contains information about version and processor specifics for the calling program to make an intelligent decision about load/noload.

There can be no ambiguous updates. The BIOS refuses to allow multiple updates for the same CPUID to exist at the same time. The BIOS also refuses to load an update for a processor that does not exist in the system.

The calling application must implement a verify function that is run after the update write function successfully completes. This function reads back the update and verifies that the BIOS returned an image identical to the one that was written. The following pseudo-code represents a calling program.

INT 15 D042 Calling Program Pseudo-code

```
//
//     We must be in real mode
//
If the system is not in Real mode
then Exit
//
//     Detect the presence of Genuine Intel processor(s) that can be updated (Family,Model)
//
If no Intel processors exist that can be updated
     then Exit
//
//     Detect the presence of the Intel BIOS Update Extensions
//
If the BIOS fails the PresenceTest
then Exit
//
// If the APIC is enabled, see if any other processors are out there
//
Read APICBaseMSR
If APIC enabled {
     Send Broadcast Message to all processors except self via APIC;
     Have all processors execute CPUID and record Family, Model, Stepping
     If all processors are not updatable
          then Exit
     }
//
// Determine the number of unique update slots needed for this system
//
NumSlots = 0;
For each processor {
     If ((this is a unique processor stepping) and
     (we have an update in the database for this processor)) {
          Checksum the update from the database;
          If Checksum fails
               then Exit;
          Increment NumSlots;
          }
     }
//
// Do we have enough update slots for all CPUs?
//
If there are more unique processor steppings that update slots provided by the BIOS
     then Exit

//
// Do we need any update slots at all?  If not, then we're all done
//
```

```
If (NumSlots == 0)
     then Exit

//
//     Record updates for processors in NVRAM.
//
For (I=0; i<NumSlots; I++) {
          //
          //     Load each Update
          //
     Issue the WriteUpdate function

     If (STORAGE_FULL) returned {
          Display Error -- BIOS is not managing NVRAM appropriately
          exit
          }
     If (INVALID_REVISION) returned {
          Display Message: More recent update already loaded in NVRAM for this stepping
          continue;
          }

     If any other error returned {
          Display Diagnostic
          exit
          }
          //
          //     Verify the update was loaded correctly
          //
     Issue the ReadUpdate function

     If an error occurred {
          Display Diagnostic
          exit
          }
          //
          //     Compare the Update read to that written
          //
          if (Update read != Update written) {
          Display Diagnostic
          exit
          }
     }
//
//     Enable Update Loading, and inform user
//
Issue the ControlUpdate function with Task=Enable.
```

8.4.3 BIOS Update Functions

Table 8-2 defines the current Pentium Pro Processor BIOS Update Functions.

Table 8-2. BIOS Update Functions

BIOS Update Function	Function Number	Description	Required/Optional
Presence test	00h	Returns information about the supported functions.	Required
Write BIOS update data	01h	Writes one of the BIOS Update data areas (slots).	Required
BIOS update control	02h	Globally controls the loading of BIOS Updates.	Required
Read BIOS update data	03h	Reads one of the BIOS Update data areas (slots).	Required

8.4.4 INT 15h-based Interface

The application that implements the INT 15h-based interface is available on the reference BIOS diskette. The program that calls this interface is responsible for providing three 64-kilobyte RAM areas for BIOS use during calls to the read and write functions. These RAM scratch pads can be used by the BIOS for any purpose, but only for the duration of the function call. The calling routine places real mode segments pointing to the RAM blocks in the CX, DX and SI registers. Calls to functions in this interface must be made with a minimum of 32 kilobytes of stack available to the BIOS.

In general, the functions return with Carry Cleared and AH contains the returned status. The general return codes and other constant definitions are listed later in this chapter.

The OEM Error (AL) is provided for the OEM to return additional error information specific to the platform. If the BIOS provides no additional information about the error, the OEM Error must be set to SUCCESS. The OEM Error field is undefined if AH contains either SUCCESS (00) or NOT_IMPLEMENTED (86h). In all other cases it must be set with either SUCCESS or a value meaningful to the OEM.

The following sections give the details of the functions provided by the INT15h-based interface.

8.4.4.1 Function 00h – Presence Test

This function verifies that the BIOS has implemented the required BIOS update functions. Table 8-3 lists the parameters and return codes for the function.

Table 8-3. Parameters for the Presence Test

Input:		
AX	Function Code	0D042h
BL	Subfunction	00h - Presence Test
Output:		
CF	Carry Flag	Carry Set - Failure - AH Contains Status. Carry Clear - All return values are valid.
AH	Return Code	
AL	OEM Error	Additional OEM Information.
EBX	Signature Part 1	'INTE' - Part one of the signature.
ECX	Signature Part 2	'LPEP'- Part two of the signature.
EDX	Loader Version	Version number of the BIOS Update Loader.
SI	Update Cnt	Number of Update Blocks the system can record in NVRAM.
Return Codes:		
SUCCESS		Function completed successfully.
NOT_IMPLEMENTED		Function not implemented. See Table 8-8 for code definitions.

Description

In order to assure that the BIOS function is present, the caller must verify the Carry Flag, the Return Code, and the 64-bit signature. Each update block is exactly 2048 bytes in length. UpdateCnt reflects the number of update blocks available for storage within non-volatile RAM. UpdateCnt must return with a value greater than or equal to the number of unique processor steppings currently installed within the system.

The loader version number refers to the revision of the update loader program that is included in the system BIOS image. The source code for this loader program is provided to OEMs and BIOS vendors by Intel.

8.4.4.2 Function 01h – Write BIOS Update Data

This function integrates a new BIOS update into the BIOS storage device. Table 8-4 lists the parameters and return codes for the function.

Table 8-4. Parameters for the Write Update Data Function

Input:		
AX	Function Code	0D042h
BL	Subfunction	01h - Write Update
ES:DI	Update Address	Real Mode pointer to the Intel Update structure. This buffer is 2048 bytes in length.
CX	Scratch Pad1	Real Mode Segment address of 64 kilobytes of RAM Block.
DX	Scratch Pad2	Real Mode Segment address of 64 kilobytes of RAM Block.
SI	Scratch Pad3	Read Mode Segment address of 64 kilobytes of RAM Block.
SS:SP	Stack pointer	32 kilobytes of Stack Minimum.
Output:		
CF	Carry Flag	Carry Set - Failure - AH contains Status. Carry Clear - All return values are valid.
AH	Return Code	Status of the Call.
AL	OEM Error	Additional OEM Information.
Return Codes:		
SUCCESS		Function completed successfully.
WRITE_FAILURE		A failure because of the inability to write the storage device.
ERASE_FAILURE		A failure because of the inability to erase the storage device.
READ_FAILURE		A failure because of the inability to read the storage device.
STORAGE_FULL		The BIOS non-volatile storage area is unable to accommodate the update because all available update blocks are filled with updates that are needed for processors in the system.
INVALID_HEADER		The Update Header contains a header or loader version that is not recognized by the BIOS.
INVALID_HEADER_CS		The Update does not checksum correctly.
SECURITY_FAILURE		The Update was rejected by the processor.
INVALID_REVISION		The same or more recent revision of the update exists in the storage device. See Table 8-8 for code definitions.

Description

The BIOS is responsible for selecting an appropriate update block in the non-volatile storage for storing the new update. This BIOS is also responsible for ensuring the integrity of the information provided by the caller, including authenticating the proposed update before incorporating it into storage.

Before writing the update block into NVRAM, the BIOS must ensure that the update structure meets the following criteria in the following order:

1. The Update header version must be equal to an Update header version recognized by the BIOS.
2. The Update loader version in the update header must be equal to the Update loader version contained within the BIOS image.
3. The Update block must checksum to zero. This checksum is computed as a 32-bit summation of all 512 double words in the structure, including the header.

The BIOS selects an update block in non-volatile storage for storing the candidate update. The BIOS can select any available update block as long as it guarantees that only a single update exists for any given processor stepping in non-volatile storage. If the update block selected already contains an update, the following additional criteria apply to overwrite it:

- The Processor signature in the proposed update must be equal to the processor signature in the header of the current update in NVRAM.
- The Update Revision in the proposed update must be greater than the Update Revision in the header of the current update in NVRAM.

If no unused update blocks are available and the above criteria are not met, the BIOS can overwrite an update block for a processor stepping that is no longer present in the system. This can be done by scanning the update blocks and comparing the processor steppings, identified in the MP Specification table, to the processor steppings that currently exist in the system.

Finally, before storing the proposed update into NVRAM, the BIOS must verify the authenticity of the update via the mechanism described in the previous section. This includes loading the update into the current processor, executing the CPUID instruction, reading MSR `08Bh`, and comparing a calculated value with the Update Revision in the proposed update header for equality.

When performing the Write Update function, the BIOS must record the entire update, including the header and the update data. When writing an update, the original contents may be overwritten, assuming the above criteria have been met. It is the responsibility of the BIOS to ensure that more recent updates are not overwritten through the use of this BIOS call, and that only a single update exists within the NVRAM for any processor stepping.

Figure 8-4 shows the process the BIOS follows to choose an update block and ensure the integrity of the data when it stores the new BIOS update.

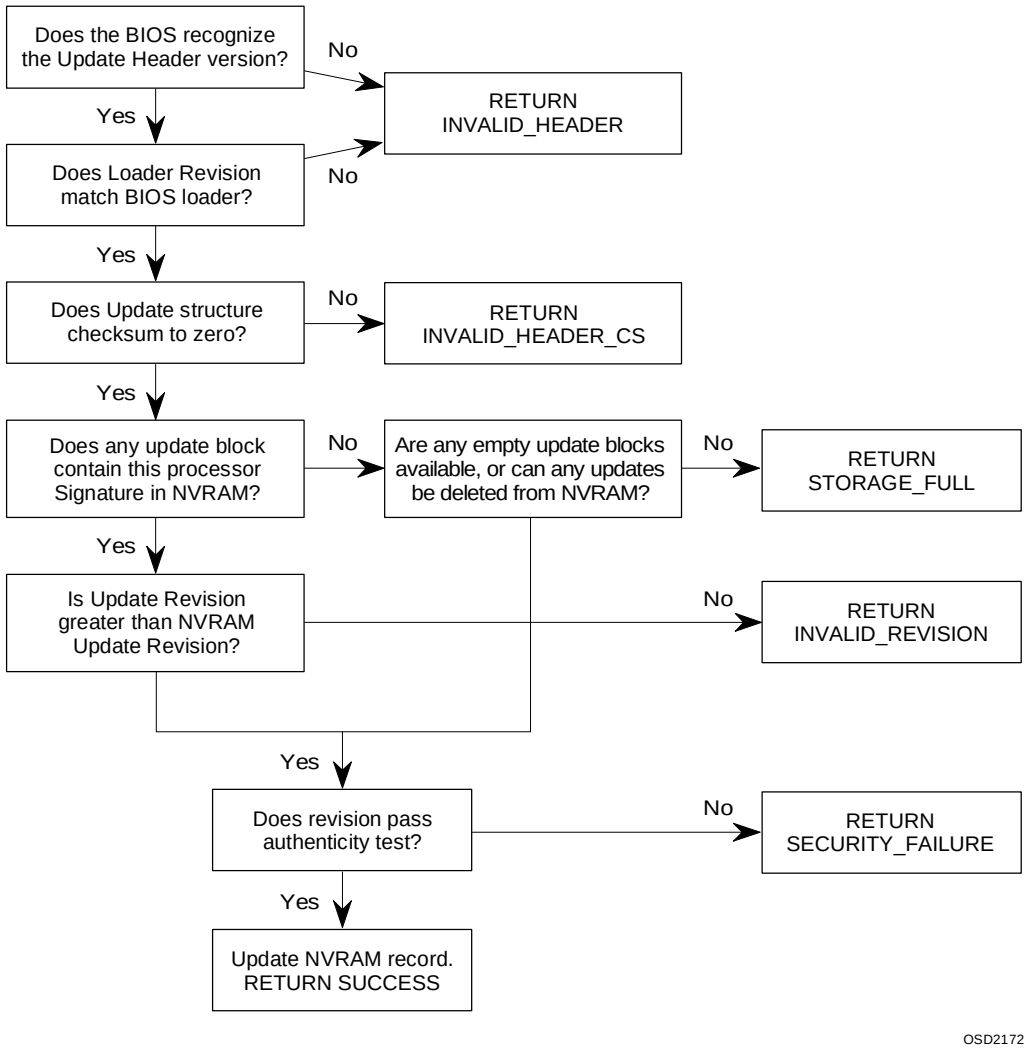

Figure 8-3. Write OperationFlow Chart

8.4.4.3 Function 02h – BIOS Update Control

This function enables loading of binary updates into the processor. Table 8-5 lists the parameters and return codes for the function.

Table 8-5. Parameters for the Control Update Subfunction

Input:		
AX	Function Code	0D042h
BL	Subfunction	02h - Control Update
BH	Task	See Description.
CX	Scratch Pad1	Real Mode Segment of 64 kilobytes of RAM Block.
DX	Scratch Pad2	Real Mode Segment of 64 kilobytes of RAM Block.
SI	Scratch Pad3	Real Mode Segment of 64 kilobytes of RAM Block.
SS:SP	Stack pointer	32 kilobytes of Stack Minimum.
Output:		
CF	Carry Flag	Carry Set - Failure - AH contains Status.
		Carry Clear - All return values are valid.
AH	Return Code	Status of the Call.
AL	OEM Error	Additional OEM Information.
BL	Update Status	Either Enable or Disable indicator.
Return Codes:		
SUCCESS		Function completed successfully.
READ_FAILURE		A failure because of the inability to read the storage device.
		See Table 8-8 for code definitions.

Description

This control is provided on a global basis for all updates and processors. The caller can determine the current status of update loading (enabled or disabled) without changing the state. The function does not allow the caller to disable loading of binary updates, as this poses a security risk. The system BIOS maintains control for disabling the loading of BIOS updates separately via the SETUP or other BIOS dependent mechanism.

The caller specifies the requested operation by placing one of the values from Table 8-6 in the BH register. After successfully completing this function the BL register contains either the Enable or the Disable designator. Note that if the function fails, the UpdateStatus return value is undefined.

Table 8-6. Mnemonic Values

Mnemonic	Value	Meaning
Enable	1	Enable the Update loading at initialization time.
Query	2	Determine the current state of the update control without changing its status.

The READ_FAILURE error code returned by this function has meaning only if the control function is implemented in the BIOS NVRAM. The state of this feature (enabled/disabled) can also be implemented using CMOS RAM bits where READ failure errors cannot not occur.

8.4.4.4 Function 03h – Read BIOS Update Data

This function reads a currently installed BIOS update from the BIOS storage into a caller-provided RAM buffer. Table 8-7 lists the parameters and return codes for the function.

Table 8-7. Parameters for Read BIOS Update Data Function

Input:		
AX	Function Code	0D042h
BL	Subfunction	03h - Read Update
ES:DI	BufferAddress	Real Mode pointer to the Intel Update structure that will be written with the binary data.
ECX	Scratch Pad1	Real Mode Segment address of 64 kilobytes of RAM Block (lower 16 bits).
ECX	Scratch Pad2	Real Mode Segment address of 64 kilobytes of RAM Block (upper 16 bits).
DX	Scratch Pad3	Real Mode Segment address of 64 kilobytes of RAM Block.
SS:SP	Stack pointer	32 kilobytes of Stack Minimum.
SI	Update Number	The index number of the update block to be read. This value is zero based and must be less than the Update Cnt returned from the Presence Test function.
Output:		
CF	Carry Flag	Carry Set - Failure - AH contains Status. Carry Clear - All return values are valid.
AH	Return Code	Status of the Call.
AL	OEM Error	Additional OEM Information.
Return Codes:		
SUCCESS		Function completed successfully.
READ_FAILURE		A failure because of the inability to read the storage device.
UPDATE_NUM_INVALID		Update Number exceeds the maximum number of update blocks implemented by the BIOS. See Table 8-8 for code definitions.

Description

The Read function enables the caller to read any update data that already exists in a BIOS and make decisions about the addition of new updates. As a result of a successful call, the BIOS copies exactly 2048 bytes into the location pointed to by ES:DI, with the contents of the Update block represented by Update Number.

An update block is considered unused and available for storing a new update if its Header Version contains the value 0FFFFFFFFh after return from this function call. The actual implementation of NVRAM storage management is not specified here and is BIOS dependent. As an example, the actual data value used to represent an empty block by the BIOS may be zero, rather than 0FFFFFFFFh. The BIOS is responsible for translating this information into the header provided by this function.

8.4.4.5 Return Codes

After the call has been made, the return codes listed in Table 8-8 are available in the AH Register.

Table 8-8. Return Code Definitions

Return Code	Value	Description
SUCCESS	00h	Function completed successfully
NOT_IMPLEMENTED	86h	Function not implemented
ERASE_FAILURE	90h	A failure because of the inability to erase the storage device
WRITE_FAILURE	91h	A failure because of the inability to write the storage device
READ_FAILURE	92h	A failure because of the inability to read the storage device
STORAGE_FULL	93h	The BIOS non-volatile storage area is unable to accommodate the update because all available update blocks are filled with updates that are needed for processors in the system
CPU_NOT_PRESENT	94h	The processor stepping does not currently exist in the system
INVALID_HEADER	95h	The Update Header contains a header or loader version that is not recognized by the BIOS
INVALID_HEADER_CS	96h	The Update does not checksum correctly
SECURITY_FAILURE	97h	The Update was rejected by the processor
INVALID_REVISION	98h	The same or more recent revision of the update exists in the storage device
UPDATE_NUM_INVALID	99h	The update number exceeds the maximum number of update blocks implemented by the BIOS

8.4.5 Protected Mode Interface

As an alternative to the INT 15h-based interface, an application can use the protected mode interface to integrate updates into the system BIOS.

8.4.5.1 Calling Convention

This section describes the method for system software to determine whether the system supports the BIOS Upgrade Protected Mode Extensions. This installation check indicates whether the system BIOS Extensions are present and the entry point to these BIOS functions.

The installation check must search for a signature of the ASCII string $IBU in system memory, starting from `0F0000h` to `0FFFFFh` at every 16-byte boundary. This signature indicates that the system supports the Pentium Pro Processor BIOS Upgrade Protected Mode Extensions. The signature identifies the start of a structure that specifies the entry point of the BIOS code implementing the support described in this document.

The system software can determine if the structure is valid by performing a **Checksum** operation: Calculate the checksum by adding up *Length* bytes from the top of the structure, including the *Checksum* field, into an 8-bit value. A resulting sum of zero indicates a valid checksum operation.

The entry points specified in this structure are the software interface to the BIOS functions. The structure element that specifies the 16-bit protected mode entry point enables the caller to construct a protected mode selector for calling this support.

Table 8-9 shows the structure of the Pentium Pro Processor BIOS Upgrade Extensions Installation Structure.

Table 8-9. BIOS Upgrade Extensions Data Fields

Field	Offset	Length	Description
Signature	00h	4 bytes	$IBU (ASCII string) where byte 0='$' (24h), byte 1= byte 1=I (49h), byte 2='B' (42h), and byte 3='U' (55h).
Version	04h	byte	01h for version 1.0. A binary value that implies a level of compliance with version changes of the Pentium Pro Processor BIOS Upgrade specification.
Length	05h	byte	17h, the length of the entire Installation Structure expressed in bytes. The length count starts at the Signature field.
Checksum	06h	byte	Calculated by adding up the number of bytes in the Installation Structure, including the Checksum field, into an 8-bit value. A resulting sum of zero indicates a valid checksum.
Real mode 16-bit offset to entry point	07h	word	Specifies the segment offset of the real mode entry point.
Real mode 16-bit code segment address	09h	word	Value varies.
16-bit protected mode offset to entry point	0Bh	word	Specifies the code segment base address, so that the caller can construct the descriptor from this segment base address before calling this support from protected mode. The offset value is the offset of the entry point. The 16-bit protected mode interface is assumed to be sufficient for 32-bit protected mode callers.
16-bit protected mode code segment base address	0Dh	dword	Value varies.
Real mode 16-bit data segment address	11h	word	Value varies.
16-bit protected mode data segment base address	13h	dword	Value varies.

The caller must also construct data descriptors for the functions that return information in the function arguments that are pointers. The only limitation is that the pointer offset can only point to the first 64 kilobytes of a segment.

If a call is made to these BIOS functions from 32-bit protected mode, the 32-bit stack is used for passing any arguments to the BIOS functions. However, it is important to note that the BIOS functions are not implemented as a full 32-bit protected mode interface. They access arguments on the stack as a 16-bit stack frame. Therefore, the caller must ensure that the function arguments are pushed onto the stack as 16-bit values and not 32-bit values. The stack parameter passing is illustrated in Figure 8-4.

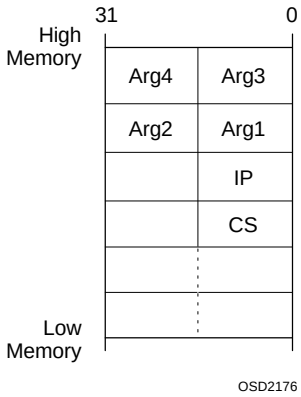

Figure 8-4. 16-bit Stack Frame on a 32-bit Stack

The system BIOS can determine whether the stack is a 32-bit stack or a 16-bit stack in 16-bit and 32-bit environments with the load access rights (LAR) byte instruction. The LAR instruction loads the high order double word for the specified descriptor. By loading the access rights for the current stack segment selector, the system BIOS can check the B-bit (Big bit) of the stack segment descriptor, which identifies the stack segment descriptor as either a 16-bit segment (B-bit clear) or a 32-bit segment (B-bit set).

In addition to executing the LAR command to get the entry point stack size, the BIOS code should avoid ADD BP, x type stack operands in runtime service code paths. These operands carry the risk of faulting if the 32-bit stack base happens to be close to the 64-kilobyte boundary. For the 16-bit protected mode interface, it is assumed that the segment limit fields will be set to 64 kilobytes. The code segment must be readable. The current I/O permission bit map must allow access to the I/O ports that the system BIOS needs to perform the function. The current privilege level (CPL) must be less than or equal to the I/O privilege level. This enables the BIOS to use sensitive instructions such as CLI and STI.

The entry point is assumed to have a function prototype of the form

```
int FAR (*entryPoint)(int Function, ...);
```

and follow the standard C calling conventions.

System software interfaces with all of the functions described in this specification by making a far call to this entry point. As noted above, the caller passes a function number and a set of arguments based on the function being called. Each function also includes an argument specifying a data selector that enables the BIOS to access and update variables within the system BIOS memory space. This data selector parameter is required for protected mode callers. The caller must create a data segment descriptor using the 16-bit protected mode data segment base address specified in the Installation Structure, a limit of 64 kilobytes, and the descriptor must be read/write capable. Real mode callers are required to set this parameter to the real mode 16-bit data segment address specified in the Installation Structure.

8.4.5.2 Function 00h – Presence Test

This function verifies that the BIOS has implemented the required BIOS update functions.

Synopsis

```
short FAR (*entryPoint)(Function, IBUSlotCount, IBUSignature, minWriteSize, IBULoaderVer,
BiosSelector);
short function;                         /* Pentium Pro Processor BIOS Upgrade Function 00h*/
unsigned short IBUSlotCount;            /* Number of BIOS Update Data Records storable by BIOS*/
char FAR *IBUSignature;                 /* Pointer to the IBUSignature*/
unsigned long minWriteSize;             /* Minimum Buffer Size to Write BIOS Update Data */

unsigned long IBULoaderVer;             /* BIOS Update Loader Code version number */
unsigned short BiosSelector;            /* BIOS readable/writable selector */
```

Description

`IBUSlotCount` Updated by the BIOS call with the total number of BIOS Upgrade Data blocks supported by the BIOS. For example, if the BIOS is capable of storing four BIOS Upgrade Data blocks, a value of 04h is returned by this function.

`IBUSignature` Updated by the BIOS call to point to the ASCII character string INTELPEP.

`MinWriteSize` Updated by the BIOS call with the size, in bytes of scratch buffer required by the BIOS to perform the Write BIOS Upgrade Data (01h) function. If the BIOS stores the BIOS Upgrade Data in a block erase device, this value would be the size of the largest block containing BIOS Upgrade Data. If the value of the `minWriteSize` returned by the BIOS is zero, then the BIOS does not require any additional buffer space to perform the Write BIOS Upgrade Data function.

`IBULoaderVer` Updated by the BIOS call with the version number of the loader code implemented by the BIOS. The initial release of the BIOS Update loader code should cause the return of `00000001h`.

`BiosSelector` Enables the system BIOS, if necessary, to update system variables contained in the system BIOS memory space. If this function is called from protected mode, the caller must create a data segment descriptor using the 16-bit protected mode data segment base address specified in the Installation Structure, a limit of 64 kilobytes, and the descriptor must be read/write capable. If this function is called from real mode, `BiosSelector` should be set to the real mode 16-bit data segment address as specified in the Installation Structure. This function is available in real mode and 16-bit protected mode.

Returns

If successful - SUCCESS, or else the Error Code is returned in AX, the FLAGS and all other registers are preserved. See Table 8-10 for return codes.

Example

The following example illustrates how the C style call interface could be made from an assembly language module:

```
push     BiosSelector

pushd    IBULoaderVer
pushd    minWriteSize
push     segment/selector of IBUSignature    ; Pointer to BIOS Update Signature
push     offset of IBUSignature
push     IBUSlotCount
push     PRESENCE_TEST
call     FAR PTR entryPoint
add      sp, 18t                             ; Clean up stack
cmp      ax, SUCCESS                         ; Function completed successfully?
jne      error
```

8.4.5.3 Function 01h – Write BIOS Update Data

This function writes the BIOS Update Data contained in the buffer specified by `IBUBuffer` into a storage area determined by the BIOS.

Synopsis

```
short FAR (*entryPoint)(Function, IBUBuffer, segStructure, BiosSelector);
short function;                          /*Pentium Pro Processor BIOS Upgrade Function 01h*/
unsigned char FAR *IBUBuffer;            /* Pointer to buffer containing BIOS Update Data */
unsigned short FAR *segStructure         /*Pointer to buffer containing Segment/Selector Info*/
unsigned short BiosSelector;             /* BIOS readable/writable selector */
```

Description

`segStructure` The segment/selector:offset of a structure containing an array of scratch segments. The sum of the size of all the segments should be at least as large as the size specified by the `minWriteSize` parameter returned from the Installation Check Function (`00h`). The structure is as follows:

```
DW          Segment/Selector 0,     Limit of Segment/Selector 0
DW          Segment/Selector 1,     Limit of Segment/Selector 1
DW          ...     ...
DW          Segment/Selector n,     Limit of Segment/Selector n
DW          NULL
```

`BiosSelector` Enables the system BIOS, if necessary, to update system variables contained in the system BIOS memory space. If this function is called from protected mode, the caller must create a data segment descriptor using the 16-bit protected mode data segment base address specified in the Installation Structure and a limit of 64 kilobytes. The descriptor must be read/write capable. If this function is called

from real mode, `BiosSelector` should be set to the real mode 16-bit data segment address as specified in the Installation Structure. This function is available in real mode and 16-bit protected mode.

The BIOS is responsible for selecting an appropriate update block within the non-volatile storage for retaining the new update. This BIOS is also responsible for ensuring the integrity of the information provided by the caller, including authentication of the proposed update before incorporating it into storage.

Before writing the update block into NVRAM, the BIOS must ensure that the update structure meets the following criteria in the order listed:

1. The Update header version must be equal to an Update header version recognized by the BIOS.
2. The Update loader version in the Update header must be equal to the Update loader version contained within the BIOS image.
3. The Update block must checksum to zero. This checksum is computed as a 32-bit summation of all 512 double words in the structure, including the header.

The BIOS selects an update block from within non-volatile storage for retaining the candidate update. The BIOS may select any available update block as long as it guarantees that only a single update exists for any given processor stepping in non-volatile storage. If the update block selected currently contains an update, then the following additional criteria apply to overwrite it:

1. The processor signature in the proposed update must be equal to the processor signature in the header of the current update in NVRAM.
2. The Update Revision in the proposed update must be greater than the Update Revision in the header of the current update in NVRAM.

If no unused update blocks are available and the above criteria are not met, the BIOS may overwrite an update block for a processor stepping that is no longer present in the system. This can be done by scanning the update blocks and comparing the processor steppings identified in the MP Specification table to the processor steppings currently in the system.

Finally, before storing the proposed update into NVRAM, the BIOS must verify the authenticity of the update via the mechanism described previously for update authentication.

When performing the Write Update function the BIOS must record the entire update, including the header and the update data. When writing an update the original contents may be overwritten, assuming the above criteria have been met. It is the responsibility of the BIOS to ensure that more recent updates are not overwritten through the use of this BIOS call, and that only a single update exists within the NVRAM for any processor stepping.

Returns

If successful - SUCCESS, else the Error Code is returned in AX, the FLAGS and all other registers are preserved. See Table 8-10 for return codes.

Example

The following example illustrates how the C style call interface could be made from an assembly language module:

```
push      BiosSelector
push      segment/selector of segStructure     ; Pointer to Segment Structure
push      offset of segStructure
push      segment/selector of IBUBuffer        ; Pointer to BIOS Update Data Buffer
push      offset of IBUBuffer
push      WRITE_BIOS_UPDATE_DATA
call      FAR PTR entryPoint
add       sp, 12t                               ; Clean up stack
cmp       ax, SUCCESS                          ; Function completed successfully?
jne       error
```

8.4.5.4 Function 02h – BIOS Update Control

This function enables or queries the global loading of BIOS Update Data into the processor(s).

Synopsis

```
short FAR (*entryPoint)(Function, segStructure, taskControl, BiosSelector);
short function;                    /* Pentium Pro Processor BIOS Upgrade Function 02h */
unsigned short FAR *segStructure   /* Pointer to buffer containing Segment/Selector Info */
unsigned short taskControl         /* Task to Perform */
unsigned short BiosSelector;       /* BIOS readable/writable selector */
```

Description

`segStructure` The segment/selector:offset of a structure containing an array of scratch segments. The sum of the size of all the segments should be at least as large as the size specified by the `minWriteSize` parameter returned from the Presence Test Function (`00h`). The structure is as follows:

```
DW          Segment/Selector 0,         Limit of Segment/Selector 0
DW          Segment/Selector 1,         Limit of Segment/Selector 1
DW          ...      ...
DW          Segment/Selector n,         Limit of Segment/Selector n
DW          NULL
```

`taskControl` Defines the task to be performed. The following tasks are defined:

Enable	`0000h`	Enable the BIOS Update loading at initialization time.
Query	`0001h`	Determine the current state of BIOS. Update loading without changing its state.

`BiosSelector` Enables the system BIOS, if necessary, to update system variables that are contained in the system BIOS memory space. If this function is called from protected mode, the caller must create a data segment descriptor using the 16-bit protected mode data segment base address specified in the Installation Structure, a limit of 64KB, and the descriptor must be read/write capable. If this function is called from real mode, `BiosSelector` should be set to the real mode 16-bit data segment address as specified in the Installation Structure. This function is available in real mode and 16-bit protected mode.

Returns

If successful - SUCCESS, else the Error Code are returned in AX, the FLAGS and all other registers are preserved. See Table 8-10 for return codes.

Example

The following example illustrates how the C style call interface could be made from an assembly language module:

```
push     BiosSelector
push     taskControl
push     segment/selector of segStructure     ; Pointer to Segment Structure
push     offset of segStructure
push     BIOS  UPDATE CONTROL
call     FAR PTR entryPoint
add      sp, 10t                              ; Clean up stack
cmp      ax, SUCCESS                          ; Function completed successfully?
jne      error
```

8.4.5.5 Function 03h – Read BIOS Update Data

This function reads the BIOS Update Data block indexed by the `IBUIndex` into the buffer specified by `IBUBuffer`.

NOTE. *This function is required for Pentium Pro Processor BIOS Upgrade Support.*

Synopsis

```
short FAR (*entryPoint)(Function, IBUBuffer, IBUIndex, BiosSelector);
short function;                         /* Pentium Pro Processor BIOS Upgrade Function 03h*/
unsigned char FAR *IBUBuffer;           /* Pointer to buffer to write BIOS Update Data */
unsigned short IBUIndex                 /* Index of BIOS Update Record to Read */
unsigned short BiosSelector;            /* BIOS readable/writable selector */
```

Description

`IBUBuffer` The segment/selector:offset of the buffer that the BIOS Update Data is to be read into.

`IBUIndex` The index of the BIOS Update Data to be read.

`BiosSelector` Enables the system BIOS, if necessary, to update system variables that are contained in the system BIOS memory space. If this function is called from protected mode, the caller must create a data segment descriptor using the 16-bit protected mode data segment base address specified in the Installation Structure and a limit of 64 kilobytes. The descriptor must be read/write capable. If this function is called from real mode, `BiosSelector` should be set to the real mode 16-bit data segment address as specified in the Installation Structure. This function is available in real mode and 16-bit protected mode.

Returns

If successful - SUCCESS, or else the Error Code is returned in AX, the FLAGS and all other registers are preserved. See Table 8-10 for return codes.

Example

The following example illustrates how the C style call interface could be made from an assembly language module:

```
push     BiosSelector
push     IBUIndex
push     segment/selector of IBUBuffer  ; Pointer to BIOS Update Data Buffer
push     offset of IBUBuffer
push     READ_BIOS_UPDATE_DATA
call     FAR PTR entryPoint
add      sp, 10t                        ; Clean up stack
cmp      ax, SUCCESS                    ; Function completed successfully?
jne      error
```

8.4.5.6 Return Codes

After the call has been made, the codes listed in Table 8-10 are available in the AX Register.

Table 8-10. Return Codes

Return Code	Value	Description
SUCCESS	00h	Function completed successfully
NOT_IMPLEMENTED	86h	Function not implemented
ERASE_FAILURE	90h	A failure because of the inability to erase the storage device
WRITE_FAILURE	91h	A failure because of the inability to write the storage device
READ_FAILURE	92h	A failure because of the inability to read the storage device
STORAGE_FULL	93h	The BIOS non-volatile storage area is unable to accommodate the update because all available update blocks are filled with updates that are needed for processors in the system
CPU_NOT_PRESENT	94h	The processor stepping does not currently exist in the system
INVALID_HEADER	95h	The Update Header contains a header or loader version that is not recognized by the BIOS
INVALID_HEADER_CS	96h	The Update does not checksum correctly
SECURITY_FAILURE	97h	The Update was rejected by the processor
INVALID_REVISION	98h	The same or more recent revision of the update exists in the storage device
UPDATE_NUM_INVALID	99h	The update number exceeds the maximum number of update blocks implemented by the BIOS

9
OverDrive® Processors

This chapter describes how to design a system to be upgradable to the future OverDrive® processor for Pentium Pro processor-based systems.

9.1 Intel OverDrive Processors and the CPU Signature

The BIOS must detect the type of CPU in the system and program the support hardware accordingly. In most cases, the BIOS determines the CPU type by reading the CPU signature, comparing it to known signatures, and executing the corresponding hardware initialization code when a match is found.

The CPUID instruction is used to determine several processor parameters. Following execution of the CPUID instruction, bits 12 and 13 of the EAX register (as indicated in Table 9-1) can be used to determine whether the processor is an original processor or an OverDrive processor. An OverDrive processor is present if bit 13 = 0 and bit 12 = 1.

9.2 OverDrive Processor CPUID

Following power-on reset the EDX register contains the values shown in Table 9-1.

Table 9-1. OverDrive Processor CPUID

Type [13:12]	Family [11:8]	Model [7:4]	Stepping [3:0]
1	6	3	x

9.3 Common Causes of Upgradability Problems Due to BIOS

CPU signature detection has been a common cause of upgradability problems. To prevent future upgradability problems with Pentium Pro processor-based systems, consider the following when programming or modifying the BIOS:

- Always use the CPU Signature and Feature flags to identify the processor.
- Never use software timing loops for delays.
- If an OverDrive processor is detected, report the presence of an “OverDrive Processor” to the end user.
- If an OverDrive processor is detected, don’t empirically test on-chip cache sizes or organization. OverDrive processor cache parameters differ from those of the Pentium Pro processor.

- If an OverDrive processor is detected, don’t use the Pentium Pro processor model specific registers and test registers. OverDrive processor MSRs differ from those of the Pentium Pro processor.
- Memory type range registers must be programmed as for a Pentium Pro processor.

NOTE. *Contact your BIOS vendor to ensure that the above requirements have been included.*

Appendix A
Query System Address Map

This appendix explains the special INT 15 call that Intel and Microsoft developed for use in ISA/EISA /PCI based systems. The call supplies the operating system with a clean memory map indicating address ranges that are reserved and ranges that are available in the motherboard

A.1 INT 15h, E820h - Query System Address Map

This call can be used in real mode only.

This call returns a memory map of all the installed RAM, and of physical memory ranges reserved by the BIOS. The address map is returned by making successive calls to this API, each returning one run of physical address information. Each run has a type which dictates how this run of physical address range should be treated by the operating system.

If the information returned from E820 in some way differs from INT-15 88 or INT-15 E801, the information returned from E820 supersedes the information returned from INT-15 88 or INT-15 E802. This replacement enables the BIOS to return any information that it requires from INT-15 88 or INT-15 E801 for compatibility reasons.

Table A-1 shows the input parameters for this call.

Table A-1. Input

EAX	Function Code	E820h
EBX	Continuation	Contains the continuation value to get the next run of physical memory. This is the value returned by a previous call to this routine. If this is the first call, EBX must contain zero.
ES:DI	Buffer Pointer	Pointer to an Address Range Descriptor structure that the BIOS fills in.
ECX	Buffer Size	The length in bytes of the structure passed to the BIOS. The BIOS fills in the number of bytes of the structure indicated in the ECX register, maximum, or whatever amount of the structure the BIOS implements. The minimum size that must be supported by both the BIOS and the caller is 20 bytes. Future implementations may extend this structure.
EDX	Signature	'SMAP' Used by the BIOS to verify the caller is requesting the system map information to be returned in ES:DI.

Table A-2 shows the output codes for this call.

Table A-2. Output

CF	Carry Flag	Non-Carry - indicates no error
EAX	Signature	'SMAP' - Signature to verify correct BIOS revision.
ES:DI	Buffer Pointer	Returned Address Range Descriptor pointer. Same value as on input.
ECX	Buffer Size	Number of bytes returned by the BIOS in the address range descriptor. The minimum size structure returned by the BIOS is 20 bytes.
EBX	Continuation	Contains the continuation value to get the next address descriptor. The actual significance of the continuation value is up to the discretion of the BIOS. The caller must pass the continuation value unchanged as input to the next iteration of the E820 call in order to get the next Address Range Descriptor. A return value of zero means that this is the last descriptor. NOTE: the BIOS can also indicate that the last descriptor has already been returned during previous iterations by returning a carry. The caller will ignore any other information returned by the BIOS when the carry flag is set.

Table A-3 defines the structure of the Address Range Descriptors.

Table A-3. Address Range Descriptor Structure

Offset in Bytes	Name	Description
0	BaseAddrLow	Low 32 Bits of Base Address
4	BaseAddrHigh	High 32 Bits of Base Address
8	LengthLow	Low 32 Bits of Length in Bytes
12	LengthHigh	High 32 Bits of Length in Bytes
16	Type	Address type of this range

The *BaseAddrLow* and *BaseAddrHigh* together are the 64-bit base address of this range. The base address is the physical address of the start of the range being specified.

The *LengthLow* and *LengthHigh* together are the 64-bit length of this range. The length is the physical contiguous length in bytes of a range being specified.

The *Type* field describes the usage of the described address range as defined in Table A-4 below.

Table A-4. Address Ranges in the Type Field

Value	Mnemonic	Description
1	AddressRangeMemory	This run is available RAM usable by the operating system.
2	AddressRangeReserved	This run of addresses is in use or reserved by the system and must not be used by the operating system.
Other	Undefined	Undefined - Reserved for future use. Any range of this type must be treated by the OS as if the type returned was AddressRangeReserved.

The BIOS can use the *AddressRangeReserved* address range type to block out various addresses as not suitable for use by a programmable device. Some of the reasons a BIOS would do this are:

- The address range contains system ROM.
- The address range contains RAM in use by the ROM.
- The address range is in use by a memory mapped system device.
- The address range is, for whatever reason, unsuitable for a standard device to use as a device memory space.

A.2 Assumptions and Limitations

- The BIOS returns address ranges describing base board memory and ISA or PCI memory that is contiguous with that baseboard memory.
- The BIOS does *not* return a range description for the memory mapping of PCI devices, ISA Option ROMs, and ISA Plug and Play cards because the operating system has mechanisms available to detect them.
- The BIOS returns chipset defined address holes that are not being used by devices as reserved.
- Address ranges defined for baseboard memory mapped I/O devices, such as APICs, are returned as reserved.
- All occurrences of the system BIOS are mapped as reserved, including the areas below 1 megabyte, at 16 megabytes (if present), and at end of the 4-gigabyte address space.
- Standard PC address ranges are not reported. Example video memory at `A0000h` to `BFFFFh` physical are not described by this function. The range from `E0000h` to `EFFFFh` is specific to the baseboard and is reported as it applies to that baseboard.
- All of lower memory is reported as normal memory. The operating system must handle standard RAM locations that are reserved for specific uses, such as the interrupt vector table (0:0) and the BIOS data area (40:0).

A.3 Example Address Map

This sample address map describes a machine which has 128 megabytes of RAM, 640 kilobytes of base memory and 127 megabytes of extended memory. The base memory has 639 kilobytes available for the user and 1 kilobyte for an extended BIOS data area. A 4-megabyte Linear Frame Buffer (LAB) is based at 12 megabytes. The memory hole created by the chipset is from 8 to 16 megabytes. Memory-mapped APIC devices are in the system. The I/O Unit is at `FEC00000h` and the Local Unit is at `FEE00000h`. The system BIOS is remapped to 1 gigabyte-64 kilobytes.

The 639-kilobyte endpoint of the first memory range is also the base memory size reported in the BIOS data segment at 40:13.

Table A-5 shows the memory map of a typical system.

Table A-5. Sample Memory Map

Base (Hex)	Length	Type	Description
0000 0000	639K	AddressRangeMemory	Available Base memory - typically the same value as is returned via the INT 12 function.
0009 FC00	1K	AddressRangeReserved	Memory reserved for use by the BIOS(s). This area typically includes the Extended BIOS data area.
000F 0000	64K	AddressRangeReserved	System BIOS
0010 0000	7M	AddressRangeMemory	Extended memory, which is not limited to the 64-megabyte address range.
0080 0000	4M	AddressRangeReserved	Chipset memory hole required to support the LFB mapping at 12 megabytes.
0100 0000	120M	AddressRangeMemory	Baseboard RAM relocated above a chipset memory hole.
FEC0 0000	4K	AddressRangeReserved	I/O APIC memory mapped I/O at FEC00000.
FEE0 0000	4K	AddressRangeReserved	Local APIC memory mapped I/O at FEE00000.
FFFF 0000	64K	AddressRangeReserved	Remapped System BIOS at end of address space.

A.4 Sample Operating System Usage

The following code segment illustrates the algorithm to be used when calling the Query System Address Map function. It is an implementation example and uses non-standard mechanisms.

```
E820Present = FALSE;
Reg.ebx = 0;
do {
    Reg.eax = 0xE820;
    Reg.es  = SEGMENT (&Descriptor);
    Reg.di  = OFFSET  (&Descriptor);
    Reg.ecx = sizeof  (Descriptor);
      Reg.edx = 'SMAP';

    _int( 15, regs );

    if ((Regs.eflags & EFLAG_CARRY)  ||  Regs.eax != 'SMAP') {
        break;
    }

    if (Regs.ecx < 20  ||  Reg.ecx > sizeof (Descriptor) ) {
        // bug in bios - all returned descriptors must be
        // at least 20 bytes long, and cannot be larger then
        // the input buffer.

        break;
    }
```

```
        E820Present = TRUE;
      .
      .
      .
      Add address range Descriptor.BaseAddress through
      Descriptor.BaseAddress + Descriptor.Length
      as type Descriptor.Type
      .
      .
      .

} while (Regs.ebx != 0);

if (!E820Present) {
      .
      .
      .
      call INT-15 88 and/or INT-15 E801 to obtain old style
      memory information
      .
      .
      .
}
```